

## Uvod

SQL(structured query language) je jezik koji se koristi za slanje upita ka RDBMS(relational database management system) koji skladisti podatke po relacionom modelu. Relacioni model je semanticki model za prezentovanje podataka koji se bazira iz dve grane matematike – set theory(teorija skupova) i predicate logic.

T-SQL je dijalekt SQL-a.

## SQL

SQL je ANSI i ISO standard language za queryovanje i storovanje podataka RDBMS-a.

SQL se sastoji iz nekoliko kategorija izraza:

- **DML** (data manipulation language) (select, update, insert... tipovi izraza za upit i manipulaciju podataka)
- **DDL** (data definition language) (create, alter, drop tipovi izraza, za manip. definicija podataka)
- **DCL** (data control language) (grant, revoke... tipovi izraza za manipulaciju dozvola)

Ova knjiga se fokusira na DML tipove izraza, znaci SELECT, INSERT, UPDATE, DELETE, TRUNCATE, and MERGE

Common misunderstanding is that TRUNCATE is a DDL statement, but in fact it is a DML statement

## Set Theory

*By a set we mean any collection M into a whole of definite, distinct objects “m” (which are called the “elements” of M) of our perception or of our thought.*

- Georg Cantor

//Mozes malo vise prouciti o skupovima kasnije ako te interesuje, ali osnovna skola

## Predicate logic

Koristi se za upravljanje podacima i izvršavanje upita nad podacima. PREDICATE je u sustini izraz/svojstvo koje je tacno ili nije tacno(holds, or doesn't hold)

Relacioni model se oslanja na predicate za zadrzanie logickog integriteta podataka i definisanje strukture istih. U sustini primer bi bio CONSTRAINT koji ce nad tabelom “employees” da postavi pravilo da plata mora da bude  $> 0 \Rightarrow \underline{\text{salary}} > 0$

## The relational model

Relacioni model je sematicki model za upravljanje podacima i manipulaciju koji se bazira na teoriji skupova i logici. Cilj ovog modela jeste omogucavanje konzistentne prezentacije podataka uz minimalnu ili bez imalo redundantnosti gde ne moras da se oslanjas na intuiciju jer jelte, bazira se na matematici.

Relacioni model uključuje koncepte poput “propositions, predicates, relations, tuples, attributes & more”.

### Propositions, predicates & relations

Da deo “relacioni” u izrazu “relacioni model” stoji tu na osnovu koncepta “relacija” je netacan. Zapravo potice iz matematičkog izraza relacija. U teoriji skupova, relacija je representation(predstava) skupa. U relacionom modelu, relacija je skup (related) informacija. U SQL-u taj skup se naziva tabela(not exact counterpart, but close).

Key point -> relacija jeste jedna tabela.

Operacije nad relacijama na osnovu relacione algebre rezultuju relacijama

//Relational model razlikuje relaciju od relacione promenljive but to keep things simple, necemo o tome sada. Koristicemo izraz “relacija” za oba.

Relacija se sastoji iz headinga i bodyja.

Heading jeste spisak atributa(kolona u SQL-u) gde svaka kolona ima svoj naziv i tip podatka koje ce da sadrzi.

Body jeste skup tuple-a(row u SQL-u) gde se svaki element identificuje kroz key. To KIS, reci cemo da je tabela skup redova.

Figure 1-1 shows an illustration of a relation called Employees. It compares the components of a relation in relational theory with those of a table in SQL.

Employees relation/table				Relational Theory	SQL Counterparts																																						
<b>PK</b>				Relation	Table																																						
<table border="1"> <thead> <tr> <th>empid INT</th><th>firstname VARCHAR(40)</th><th>lastname VARCHAR(40)</th><th>hiredate DATE</th></tr> </thead> <tbody> <tr><td>5</td><td>Sven</td><td>Mortensen</td><td>10/17/2021</td></tr> <tr><td>8</td><td>Maria</td><td>Cameron</td><td>3/5/2022</td></tr> <tr><td>3</td><td>Judy</td><td>Lew</td><td>4/1/2020</td></tr> <tr><td>2</td><td>Don</td><td>Funk</td><td>8/14/2020</td></tr> <tr><td>6</td><td>Paul</td><td>Suurs</td><td>10/17/2021</td></tr> <tr><td>9</td><td>Patricia</td><td>Doyle</td><td>11/15/2022</td></tr> <tr><td>1</td><td>Sara</td><td>Davis</td><td>5/1/2020</td></tr> <tr><td>4</td><td>Yael</td><td>Peled</td><td>5/3/2021</td></tr> <tr><td>7</td><td>Russell</td><td>King</td><td>1/2/2022</td></tr> </tbody> </table>				empid INT	firstname VARCHAR(40)	lastname VARCHAR(40)	hiredate DATE	5	Sven	Mortensen	10/17/2021	8	Maria	Cameron	3/5/2022	3	Judy	Lew	4/1/2020	2	Don	Funk	8/14/2020	6	Paul	Suurs	10/17/2021	9	Patricia	Doyle	11/15/2022	1	Sara	Davis	5/1/2020	4	Yael	Peled	5/3/2021	7	Russell	King	1/2/2022
empid INT	firstname VARCHAR(40)	lastname VARCHAR(40)	hiredate DATE																																								
5	Sven	Mortensen	10/17/2021																																								
8	Maria	Cameron	3/5/2022																																								
3	Judy	Lew	4/1/2020																																								
2	Don	Funk	8/14/2020																																								
6	Paul	Suurs	10/17/2021																																								
9	Patricia	Doyle	11/15/2022																																								
1	Sara	Davis	5/1/2020																																								
4	Yael	Peled	5/3/2021																																								
7	Russell	King	1/2/2022																																								
				set of attributes	set of columns																																						
				set of tuples	multiset of rows																																						

FIGURE 1-1 Illustration of Employees relation

Keep in mind da elementi u skupu(tuples) i kolone/atributi mozda mogu izgledati kao da imaju redosled. Oni nemaju redosled(order) i ne mogu imati redosled, po definiciji skupova.

To isto vazi i za recorde unutar tabele.

Kada formalizujes neku pozavu u prirodu tako da je mozes prezentovati kroz skup – ti ces u sustini koristiti razne predicates. Za opisivanje kolone/atributa, npr ti ces koristiti TIP(type).

**TYPE** je najjednostavniji oblik predikata. Npr int ogranicava kolonu da vrednost bude u opsegu -2,147,483,648 do 2,147,483,647. Tip nije samo integer/date. To su jednostavnii tipovi.

Tip moze biti i kompleksan gde bi kompleksan bio npr neka enumeracija vrednosti.

## Missing values

Jelite, postoji debata o tome da li bi predikat trebao podrzavati dvo-vrednostne ili trovrednosne vrednosti. Dvovrednosni predicate bi bio npr true/false.

Trovrednosni bi bio true/false/null tj nepostojanje vrednosti. Dvovrednosni sistem podrazumeva “the law of excluded middle” sto je matematicko pravilo koje kaze da “nema sredine”. Ili jesi, ili nisi. Sa Trovrednosnim sistemom ti kazes “unknown/ne znam” da je takodje vrednost.

Neki tvrde da se NULL/unknown i trovrednosni predicate logic na ubrajaju u relacioni model.

SQL supports three value predicate logic.

Postoji "Codd" koji tvrdi da treba da postoji cetvorovrednosni predicate sistem u kom kada postoji nepoznata vrednost(null), ona se deli na "primenjivu" i "neprimenjivu".

Neprimenjiva bi bila kada employee nema mobilni telefon pa onda postojanje atributa mobilnog telefona nema primenu tj non-applicable.

Ukoliko employee ima mobilni telefon onda je "primenjiva" nepoznata vrednost jer jelte, ima telefon samo mi ne znamo koji je. SQL ne podrzava cetvorovrednosni predicate sistem.

## Constraints

Jedna od prednosti relacionog modela jeste mogucnost za definisanjem integriteta podataka kao deo modela. Constraints su pravila koja se jelte definisu u modelu, a koje sprovodi RDBMS. Jedan od najcescijih jeste da li atribut dozvoljava ili ne dozvoljava null marker.

Constraints takodje se namecu kroz sam model – npr ako imas tabelu sa 3 kolone, onda neneses imati 4 atributa :)

Za sprovodjenje referential integrity koriste se candidate keys i foreign keys. Na osnovu candidate key mozes uniquely identifikovati record u tabeli npr. Tipicno je da se jedan od candidate keys koristi kao primary key.

U slucaju ljudi to ce biti JMBG npr, a mozes i "id". Svi drugi candidate keys pored tvog primary key-a se nazivaju alternate keys.

Foreign keys are used to enforce referential integrity. Definise se nad jedan ili vise atributa u relaciji i referencira candidate key u toj relaciji ili nekoj drugoj relaciji.

## Normalization

Relacioni model poseduje pravila normalizacije, kao i u matematici. Naime, sluzi da se smanji redundantnost podataka.

Postoje x3 nominalne forme - 1NF, 2NF, 3NF

1<sup>st</sup> nominal form(1NF) kaze da svaki record(tuple) u tabeli(relaciji) mora biti unique i da svaka kolona(atribut) mora biti atomican.

2<sup>nd</sup> nominal form(2NF) ima dva pravila. Prvo kaze da podaci moraju da zadrze njihov prvu normalnu formu. Drugo je vise vezano za kljuceve(candidate & foreign).

Svaki atribut u relaciji sa candidate key, teba da bude funkcionalno zavisan od tog candidate key-a. To znaci da atribut ne moze da bude parcijalno zavisan od candidate key-a.

To put it more informally, if you need to obtain any nonkey attribute value, you need to provide the values of all attributes of a candidate key from the same tuple

Orders	
PK	<u>orderid</u>
PK	<u>productid</u>
	orderdate
	qty
	customerid
	companynamne

FIGURE 1-2 Data model before applying 2NF

The second normal form is violated in Figure 1-2 because there are nonkey attributes that depend on only part of a candidate key (the primary key, in this example). For example, you can find the *orderdate* of an order, as well as *customerid* and *companynamne*, based on the *orderid* alone.

To conform to the second normal form, you would need to split your original relation into two relations: *Orders* and *OrderDetails* (as shown in Figure 1-3). The *Orders* relation would include the attributes *orderid*, *orderdate*, *customerid*, and *companynamne*, with the primary key defined on *orderid*. The *OrderDetails* relation would include the attributes *orderid*, *productid*, and *qty*, with the primary key defined on *orderid* and *productid*.

Orders		OrderDetails	
PK	<u>orderid</u>	PK,FK1	<u>orderid</u>
PK		PK	<u>productid</u>
	orderdate		
	customerid		
	companynamne		qty

FIGURE 1-3 Data model after applying 2NF and before 3NF

TLDR ,ne mozes da imas dva primary key-a gde ces na osnovu jednog ili na osnovu drugog raditi pretrage.

3<sup>rd</sup> nominal form(3NF) takodje ima dva pravila. Prvo je da mora se ispostovati 2<sup>nd</sup> nominal form. Drugo je da svi non-key atributi moraju biti individualni i nezavisni jedni od drugih(must be dependent on candidate keys nontransitively.)

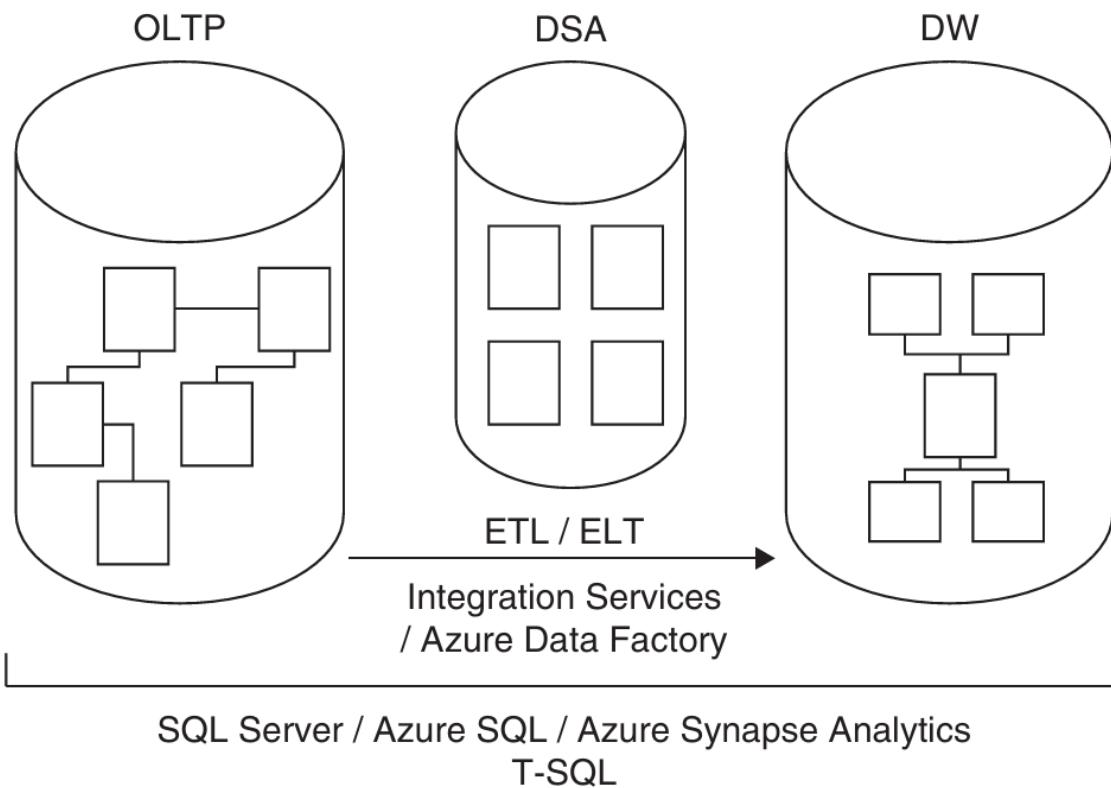


**Note** SQL, as well as T-SQL, permits violating all the normal forms in real tables. It's the data modeler's prerogative and responsibility to design a normalized model.

## Types of database workloads

Dva najcesca workloada su za OLTP(Online transaction processing) i DW(data warehousing).

Za DW to mozes sa AzureSQL/Sq|Server koji koristi SMP(symmetric multiprocessing) architecture ILI za malo zahtevnije workloade Synapse Azure Analytics koji koristi MPP(Massive parallel processing)



**FIGURE 1-5** Classes of database systems

Here's a quick description of what each acronym represents:

- OLTP: online transactional processing
- DSA: data-staging area
- DW: data warehouse
- ETL / ELT: extract, transform, and load; or extract, load, and transform

## OLTP – Online Transaction Processing

Relacioni model jeste primarno “optimizovan” za OLTP workloads koji predstavljaju neki bazicni CRUD kada ti imas online transakcije koje na taj nacin rade sa podacima, znaci primarni cilj ti je data entry bez da imas neki reporting.

Ovde su tabele cesto i normalizovane, ali upravo zbog te normalizacije ovaj use case ne odgovara reportingu jer u OLTP mi cemo imati puno entiteta sa kompleksnim relationshipima.

Ako bi to pokusali na reporting scenariu, mi cemo imati mnogo kompleksnih queryja sa losim performansom.

## DW – Data Warehousing

Ovo je okruzenje koje je optimizovano za pribavljanje podataka i reporting purposes. Ukoliko opsluzuje celu organizaciju onda se to naziva data warehouse. Ako opsluzuje samo departman unutar organizacije ili jedan mali specifican context onda se naziva data mart.

U ovim slucajevima model podataka je optimizovan primarno za retrieval needs.

The model has intentional redundancy, fewer tables, and simpler relationships, ultimately resulting in simpler and more efficient queries than an OLTP environment.

Najjednostavniji data warehouse dizajn se naziva star schema. Ona ukljuce nekoliko dimenzija schema i fact table(tabelu cinjenica). Svaka dimenzija predstavlja jednu temu na osnovu koje zelis da radis reportove – na primer sistem koji radi sa “orders & sales” verovatno ce zahtevati sledece dimenzije: customers, products, employees & time.

Svaka dimenzija je implementirana kao jedna tabela sa redundantnim podacima.

single ProductDim(product dimension) table instead of three normalized tables: Products, ProductSubCategories, and ProductCategories

Ukoliko uzmes i normalizujes tabelu dimenzije(u vise tabela jelte), onda dobijas nesto sto se naziva snowflake dimension. Takva schema koja podrzava normalizovane dimenzije se naziva snowflake schema(znaci imas jedan ili vise snowflake, ne mora sve u snowflake).

**Fact table** tj tabela cinjenica sadrzi “facts and measures”(cinjenice i mere) poput kolicina, vrednost za svaku relevantnu kombinaciju kljuceva dimenzija.

Npr za svaku relevantnu kombinaciju customer, product, employee & day – tabela cinjenica ce imati red koji sadrzi quantity & value.

Note – podaci u dimenzijama su cesto “pre agregirani” za razliku od OLTP podataka koji su skladisteni cesto “po transakciji”.

## ETL/ELT

Proces koji povlaci podatke sa izvornih sistema(OLTP & drugi), manipulise im i ucitava ih u datat warehouse se naziva **Extract, transform & load ETL**.

Neki sistemi prvo povlaze podatke sa izvornih sistema, ucitavaju ih u data warehouse i na kraju ih transformisu se nazivaju **Extract Load Transform – ELT**

**Microsoft provides an on-premises tool called Microsoft SQL Server Integration Services (SSIS) to handle ETL/ELT needs which comes with SQL Server license.**

Microsoft also provides a serverless cloud service for ETL/ELT solutions called **Azure Data Factory**

Cesto procesi koji vrse OLTP -> DW i koji sadrže neki ETL/ELT posao, imaju i DSA(**Data-Staging-Area**) koja može biti u nekoj npr relacionoj bazi i sluzi kao data cleansing area. Zona gde ce se podaci “preciscavati”.

## SQL Server Architecture

### On-Prem & Cloud flavors of Sql Server.

Inicijalno je tu bio samo jedan RDBMS – Microsoft Sql Server koji je sluzio za on premises baze. Danas unutar svoje “data platforme”, MSFT nudi onprem, box solutions i service based cloud solutions. Mnogo vise :)

### On Premises

Ova nijansa RDBMS-a se naziva SQL Server koji je najtradicionalnija verzija sto msft nudi. Ovde customer vodi racuna o svemu – hardver, patchovi, disaster recovery, data retention i slicno.

Customer moze da instalira i vise instanci SQL servera na istom backendu i radi sta hoce u sustini. Max kontrola.

## Cloud

Cloud computing varijanta nudi cloud compute & cloud storage opcije “po zahtevu” sa deljenog “pool of resources”. MSFT RDBMS mogu biti ponudjene i sa public cloud i private cloud varijanti.

Za private cloud infrastrukturu se misli na cloud infrastrukturu koju jedna organizacija koristi putem tehnologije virtualizacije. Cesto je hostovana lokalno.

Za public cloud infrastrukturu – ovde microsoft sve resava za tebe tako sto svi servisi public cloud infrastrukture su ti dostupni preko “network” tj mreze.

Ovde imamo x2 opcije, a to su IaaS i PaaS.

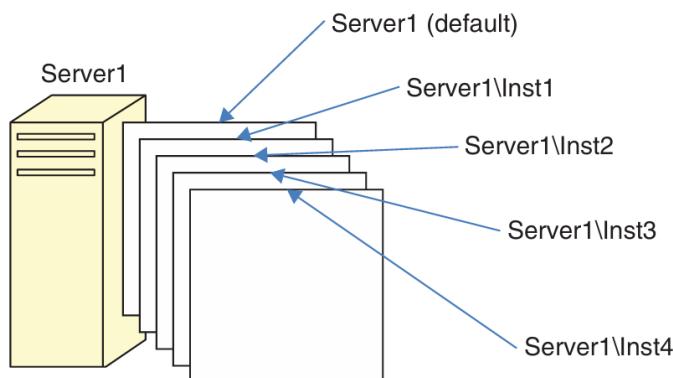
**IaaS** je kada hostujes SQL Server na VM. Ovde je na tebi da podesavas sve osim underlying hardware, jel tako. It’s essentially like maintaining your own SQL Server installation—one that happens to reside on Microsoft’s hardware.

**PaaS** je ukoliko koristimo microsoft managed sql server instance. Hardware, software installation and maintenance, high availability and disaster recovery, and patching are all responsibilities of Microsoft.

Customer jedino o cemu treba da brine je index & query tuning.

## SQL Server Instances

U sustini ovo su instalacije SQL Servera na jednom on prem servisu. Svaka instalacija(proces) ne deli nista drugo izmedju sebe osim hardvera. Kompletno je nezavisna.



**FIGURE 1-6** Multiple instances of SQL Server on the same computer

Kada imas vise instanci na istom serveru, samo jedna instance moze biti default instance. Sve ostale moraju biti named instances. O tome koja instance je default, odluce se na kreiranju same instance i ne moze se kasnije promeniti.

Da bi se client povezao na instancu:

**(default) – {pc\_ip/pc\_name} (Server1)**

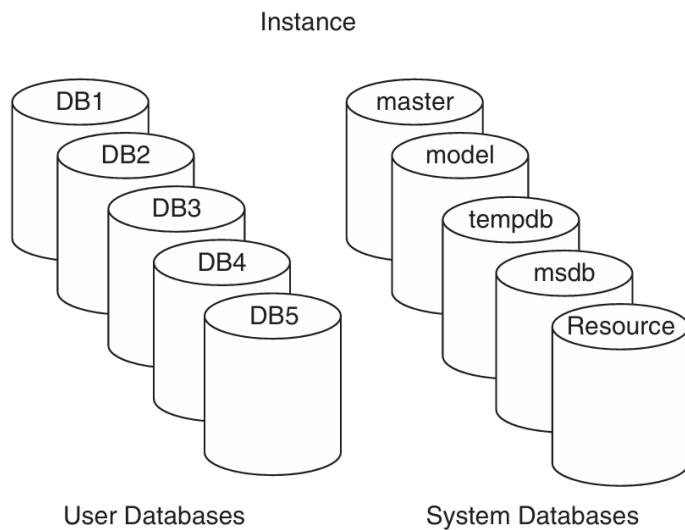
**(named\_instance) – {pc\_ip/pc\_name}\{instance\_name} (Server1\Inst2)**

## Databases

Baze(databases) su kao containeri koji sadrze razlicite objekte poput tabela, viewova, stored procedura i drugih objekata. Svaka instance SQL servera moze sadrzati vise baza.

Setup proces takodje automatski postavi nekoliko sistemskih baza za interne mssql potrebe.

Ti naravno mozes postaviti svoje user databases koje ce sadrzati podatke koje ti zelis.



Sistemske baze:

- **master** – Sadrzi “instance wide” metadata informacije, konfiguracije servera, informacije o svim bazama unutar instance i informacije inicijalizacije.
- **model** – Ova baza se koristi kao template za nove baze. Svaka nova baza koju kreiras incijalno je kreirana kao kopija model baze, pa ako zelis neke objekte poput user defined data types da budu dostupni u svim novokreiranim bazama ili da novokreirane baze budu unapred konfigurisane na neki nacin, to ovde mozes da podesis. Note, dodatne izmene u “model” bazi se nece odraziti na vec postojece baze

- **tempdb** – ovde SQL Server skladisti privremene podatke poput work tables, sort & hash table data kada treba da se persistuju, row versioning information i slicno. SQL Server ti dozvoljava da ovde kreiras i svoje tabele, ali **tempdb** se unistava i re-kreira kao kopija **model** baze svaki put kada restartujes instancu SQL servera.
- **msdb** – Ovu bazu koristi najcesce SQL Server Agent servis da skladisti svoje podatke. Njegov zadatak je automacija u koje spadaju “jobs, schedules, and alerts”. SQL Server Agent takodje vrsi replikacije. Pored toga, msdb sadrzi podatke o SQL Server features poput database mail, service broker, backups, and more.
- **resource** – resource baza je skrivena, read only db koja cuva definicije o svim sistemskim objektima. Kada queryujes system objekte u bazi, ispada kao da su oni u **sys schema** lokalne baze, ali u stvarnosti sve te definicije se nalaze u resource bazi.

Mozes kreirati 32.767 user baza unutar instance. User database skladisti objekte i podatke aplikacije.

Postavljanjem “collation” propertya na db level ce ti odluciti o default language support, case sensitivity & sort order za character data u bazi. Ako ne specificiras, uzimaju se default collation instance.

#### ■ Security concepts

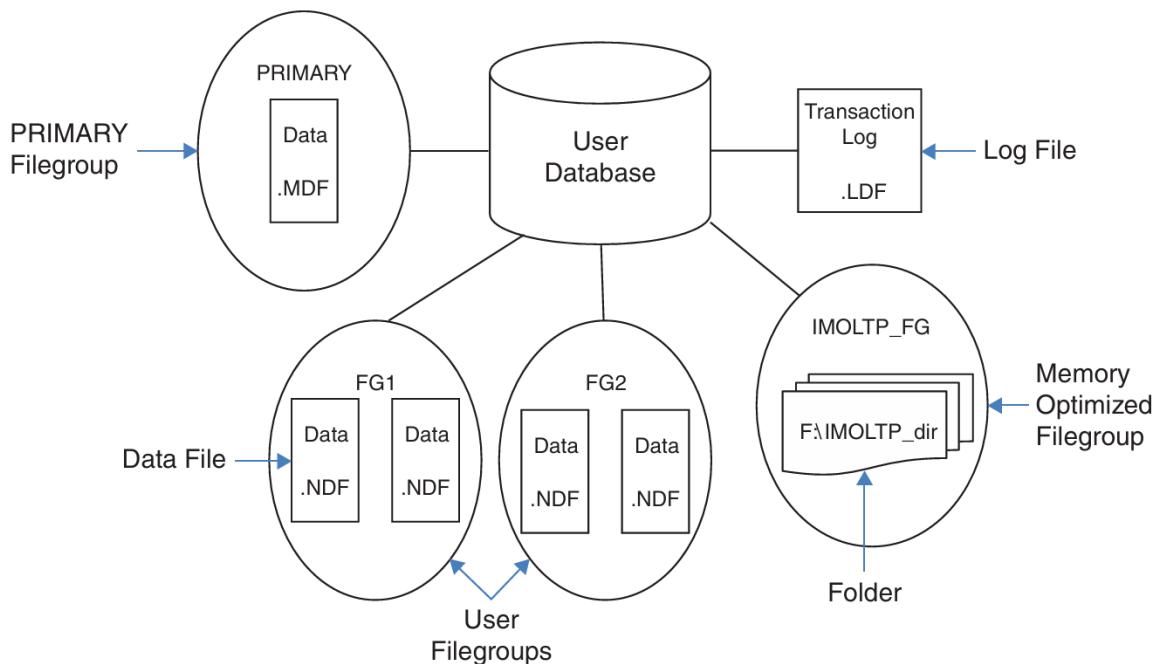
Naime kako bi pristupio instanci moras imati pristup ti kao user. To je login.

Potom kada zelis da pristupis nekoj bazi, db admin mora da namapira tvoj login na database usera.

Database user je entitet na kojeg se nadodaju permisije ka db objektima. SQL Server supportuje **contained databases** feature koji “breaks the connection” izmedju db usera i instance level logina. User(Windows/SQL Authenticated) je fully contained unutar odredjene baze i nije povezan u login na instance level.

To znaci da kada se povezujes na SQL Server kao user, moras da specificira bazu na koju se povezujes. Ne mozes kasnije switchovati baze.

Kada radis sa Azure SQL Managed instance, ti se brines jedino o logickom sloju. Ako radis sa SQL Serverom na VM ili on-prem, onda moras da radis i sa fizickim slojem baze. To znaci da radis i sa sledecim:



**FIGURE 1-8** Database layout

Baza se sastoji iz data files, transaction log files, opciono checkpoint files koji sadrže memory optimized data(In-Memory OLTP feature).

Svaka baza mora da ima barem jedan data file i barem jedan log file. Data files cuvaju object data, a log files informacije koje SQL Serveru trebaju da bi odrzao transakcije.

Pri kreiranju baze mozes definisati razlicita svojstva za data i log files ukljucujuci i file name, location, initial size, max size i autogrowth increment.

Iako SQL Server moze da upisuje u vise data files u paraleli, sa log fileovima moze samo jedan(one at a time) u sekvencialnom pristupu.

**Datafiles** su organizovani u **filegroups** koje su u sustini logicke grupacije. Filegroup je target za kreiranje objekta poput table/index. Informacije o objektu ce biti rasporedjene kroz fileove koji pripadaju target file groupi.

Filegroups su tvoj nacin za kontrolisanje fizickih lokacija tvojih objekata. Database mora posedovati barem jednu filegroup koja se naziva **PRIMARY** i opciono moze imati druge filegroups.

Primarni data file(**.mdf** ekstenzija) se nalazi u primary filegroup i to je pirmarni data file za bazu i system catalog. Opciono mozes dodati secondary data files(**.ndf** ekstenzija) unutar PRIMARY.

User filegroups sadrže samo secondary data files. Možes odabrati koja filegroup je marked kao default filegroup.

Kada object creation statement ne govori preciznije o tome gde se kreira, on će se kreirati u default filegroup.

.mdf -> master data file

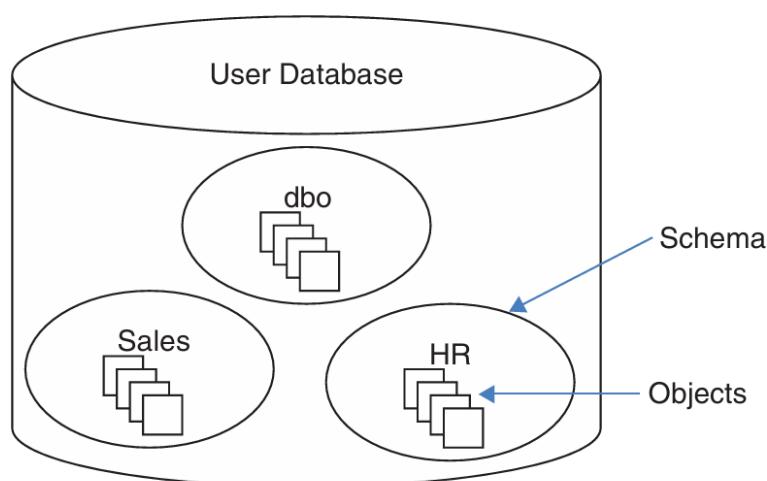
.ldf -> log data file

.ndf -> not master data file

SQL Server db engine u sebi ima takođe i memory-optimized engine “In-Memory OLT”. Uz taj feature možes integrisati memory optimized objects poput memory optimized tables i natively compiled modules(procedures, functions & triggers) u bazu. Da bi to uradio moras da kreiras filegroup u bazi koji će markovati kao containing memory optimized data i unutar njega bar jedan path u folder. Sql server storeuje checkpoint files u memory optimized data u tom folderu i koristi ih da recoveruje data svaki put kad je sql server restartovan.

## Schemas & Objects

Da je baza container objekata je malo simplified verzija. Baza sadrži schemas. Schemas sadrži objekte. Schema je container(namespace) objekata poput tabela, views, stored procedures & drugo.



**FIGURE 1-9** A database, schemas, and database objects

Permisije mozes kontrolisati na schema nivou. Npr useru mozes dozvoliti SELECT permisija za samo jednu schemu.

Pri svojim upitima ukoliko ne ukljucis schema name, SQL Server ce apply process za resolve schema name poput proveravanja da li objekt postoji u **user default schema**, a ako ne postoji onda proverava da li postoji u **dbo** schema.

Ukoliko isti objekat postoji u vise schema onda je bolje da specificiras schema pre nego pokusas da ga referujes jer mozda dobijes razlicite rezultate

## Creating tables and defining data integrity

Ovi primeri ovde se odnose na **dbo** schema koja se automatski kreira u svakoj bazi i postavlja za default schema za usere osim ukoliko nije drugacije naznaceno.

USE TSQLV6; --> sets current db context

DROP TABLE IF EXISTS dbo.Employees; --> mozes i da ni ne kreiras ako postoji vec  
CREATE TABLE dbo.Employees --> **dbo.Employees** two part name(msft recommended)  
(  
empid INT NOT NULL,  
firstname VARCHAR(30) NOT NULL,  
lastname VARCHAR(30) NOT NULL,  
hiredate DATE NOT NULL,  
mgrid INT NULL,  
ssn VARCHAR(20) NOT NULL,  
salary MONEY NOT NULL  
);

Ako ommitujes schema name onda ide u default schema.

Pri create table specificiras <attribute\_name> <data\_type> <>nullability>

## Defining data integrity

Imamo procedural data integrity gde se integritet podataka cuva kroz stored procedure i trigere i declarative data integrity gde se integritet cuva kroz samu definiciju modela.

declarative data integrity: primary key, unique, foreign key, check, default constraints, nullability, data type.

Takve constraintove mozes u toku CREATE TABLE definisati, a i dodati kasnije kroz ALTER TABLE. Svi constraintovi osim default constrainta se mogu definisati kao composite constraints.

## Primary key constraints

PK Constraint forsira uniqueness of rows i nedozvoljava NULL u constraint atributima.

Svaka kombinacija vrednosti atributa se znaci moze pojaviti samo jednom(unique).

Ako pokusas da dozvolis null pod PK, dobijes error.

```
ALTER TABLE dbo.Employees  
ADD CONSTRAINT PK_Employees  
PRIMARY KEY(employeedid)
```

Behind the scenes, sql server kreira index da bi naterao uniqueness logickog primary key constrainta. Unique index je fizicki objekat kojeg sql server koristi da enforceuje uniqueness.

Indexi takodje se koriste da bi se speed up queries da bi se izbegao sorting ili nepotrebni full table scans.

## Unique constraints

Enforceuje uniqueness row-a. Dozvoljava ti implementaciju koncepta alternate key-a.

Mozes imati vise unique constraintova na atributima tabele. Kolona moze biti i null, nije restricted samo na not null kolone.

```
ALTER TABLE dbo.Employees  
ADD CONSTRAINT UNQ_Employees_ssn  
UNIQUE(ssn)
```

Behind the scenes kreira se isto unique index.

Jedna interesantna stvar, po SQL-u unique se definise kao "Unique constraint na T je zadovoljena akko ne postoje dva reda R1 i R2 od T takvih da R1 i R2 imaju istu ne null vrednost".

Znaci unique se ubraja samo za vrednosti. Kako je null nepostojanje vrednosti, mi trebali imati vise redova sa unique attributom gde je null "vrednost".

To je kako SQL standard definise null handling unique constraintova.

Po SQL Serveru ti ako stavis unique na nullable atribut mozes imati samo jedan null od svih redova.

Ukoliko zelis unique po SQL standardu:

```
CREATE UNIQUE INDEX idx_ssn_notnull ON dbo.Employees(ssn) WHERE ssn IS NOT NULL
```

## Foreign key constraints

Foreign key razresafa referentni integritet. Poenta FK je da ogranici vrednosti u FK koloni na one koje postoje u referenced columns.

```
CREATE TABLE dbo.Odrders
(
    orderid INT NOT NULL,
    custid INT NOT NULL,
    desc VARCHAR(20),
    qty INT NOT NULL,
    CONSTRAINT PK_Orders
        PRIMARY KEY(orderid)
)
```

I sad da ubacis FK:

```
ALTER TABLE dbo.Orders
    ADD CONSTRAINT FK_Orders_Employees
        FOREIGN KEY(empid) REFERENCES dbo.Employees(empid)
```

Postoje “referencing” i “referenced” tabele u FK. Referencing je one koje pozivaju referenced tabelu. Referencing je orders. Referenced je employees.

Nulls su dozvoljeni u FK po defaultu. Cak iako PK ne dozvoljava NULL.

Ovako gore opisani FK je samo za enforceovanje referential action-a koji se naziva “NO ACTION”.

To znači da ukoliko pokusas da obrises row iz referenced table ili update referenced candidate key attributes, to će biti rejected ako related rows postoje u referencing tabelama.

Mozes definisati FK sa actions.

Postoje opcije ON DELETE i ON UPDATE sa akcijama poput CASCADE, SET DEFAULT i SET NULL.

CASCADE – operacije će se cascadeovati u related rows. ON DELETE CASCADE znači da će se i delete preneti.

SET DELETE i SET NULL – operacije koje u referenced tabeli menjaju candidate key atribut, postavice vrednost u referencing tabelama na default/null vrednost. Ovde ce postojati potom “orphan rows”

Parent rows with no related child rows su ovek allowed.

## CHECK Constraints

Definise predicate koji row mora da ispostuje da bi bio dodat/modifikovan.

```
ALTER TABLE dbo.Employees
```

```
ADD CONSTRAINT CHK_Employees_salary  
CHECK(salary > 0.00);
```

Modifikacije se priznaje samo kada je evaluacije TRUE ili UNKNOWN. Npr salary od -1000 ce biti rejected, ali 50000 i NULL bice accepted. Ako je null dozvoljen.

## Default constraint

Vezuje se za neki konkretni atribut. Koristi se kako bi osigurao vrednost nekog atributa ukoliko ne bude bio prosledjen pri kreiranju reda.

```
ALTER TABLE dbo.Orders
```

```
ADD CONSTRAINT DFT_Orders_orderts  
DEFAULT(SYSDATETIME()) FOR orderts;
```

Default expression invokeuje SYSDATETIME fn koja uzvraca sa current date i time vrednosti.

## Single Table Queries

### SELECT Statement

Ovaj statement sluzi za vršenje upita nad tabelom, primenjivanje logickih manipulacija i vracanje rezultata. Ovde ce biti opisan i logicki redosled po kom se razliciti query upiti procesiraju(logical query processing).

DB Engine ima svoj query optimizer koji cesto dodaje dosta transformacija i precica u “fizickom procesiranju queryja” kao deo optimizacije.

```
SELECT empid, YEAR(orderdate) as orderyear, COUNT(*) AS numorders  
FROM orders  
WHERE custid = 71  
GROUP BY empid, YEAR(orderdate)  
HAVING COUNT(*) > 1  
ORDER BY empid, orderyear
```

U vecini programskih jezika bi se stvari izvrsavale redosledom kako ih ispises, ali u SQL je malo drugacije. U gore queryju je redosled sledeci(logical order):

1. FROM (queries the rows from the orders table)
2. WHERE (filters only orders where custid is 71)
3. GROUP BY (groups the orders by employee id and orderdate)
4. HAVING ( filters only groups(emp id and ord yr) having more than one order)
5. SELECT ( returns emp id, ord yr & num of orders for each group)
6. ORDER BY (sorts the rows in the output)

Iako sintaksno ide prvo “select”, on se skoro poslednji izvrsava

## FROM clause

U ovom clause specificiras imena tabela nad kojima zelis da vrsis upit.

### FROM Sales.Orders

Kada selektujes sve recorde iz tabele, ne moras da delimitujes identificere(kolone).

Ovako nesto: **SELECT id, name, surname FROM users** je sasvim okej.

Medjutim ukoliko je tvoja kolona “reserved keyword”, pocinje s brojem, sadrzi razmak i slicno – onda koristis T-SQL specific formu za delimiting ili navodnike:

**SELECT id, “Order Details”, [Full Name] from [dbo].[Users]**

## Where Clause

Definises predikat ili logicki izraz na osnovu kog se filtriraju rezultati iz FROM clause-a. Oni redovi koji su evaluirani sa “true” su vraci, a ostali odbaci. Pod ostali to su “FALSE” i “UNKNOWN”. Zapamti – three-valued predicate logic.

WHERE izraz moze mnogo da utice na performans, na primer ako u query-u uzmes izraz koji koristi indexe, onda ces mnogo brze filtrirati rezultate nego da se radi full table scan.

## GROUP BY Clause

Mozes koristiti GROUP BY kako bi logicki rasporedio redove u grupe. Grupe su odredjene na osnovu elemenata/izraza u samom GROUP BY-u.

Npr: **GROUP BY empid, YEAR(orderdate)**

Ovaj gore izraz znaci da ce se u group by phase kreirati grupa za po svaku distinct vrednost empid i godine.

Ukoliko koristis GROUP BY, onda se izrazi SELECT, HAVING i ORDER BY odnose na grupe, a ne individualne redove.

Elements that do not participate in the GROUP BY clause are allowed only as inputs to an aggregate function such as COUNT, SUM, AVG, MIN, or MAX

```
SELECT  
    empid,  
    YEAR(orderdate) AS orderyear,  
    SUM(freight) AS totalfreight,  
    COUNT(*) AS numorders  
FROM Sales.Orders  
WHERE custid = 71  
GROUP BY empid, YEAR(orderdate);
```

This query generates the following output:

empid	orderyear	totalfreight	numorders
1	2020	126.56	1
2	2020	89.16	1
9	2020	214.27	1
1	2021	711.13	2
2	2021	352.69	1
3	2021	297.65	2
4	2021	86.53	1
5	2021	277.14	3
6	2021	628.31	3
7	2021	388.98	1
8	2021	371.07	4
1	2022	357.44	3
2	2022	672.16	2
4	2022	651.83	3
6	2022	227.22	1
7	2022	1231.56	2

(16 rows affected)

Count npr vraca broj redova u svakoj grupi.

Sum npr vraca sumu svih freight values unutar grupe.

NOTE: Sve aggregate funkcije ignorisu "NULL" tj nepostojanje vrednosti

Na primer da imamo grupu sa redovima u kojima je jedna kolona null(npr da imamo 5 redova u grupi), onda ce COUNT(\*) vratiti 5.

Ukoliko radimo npr COUNT(by\_col\_with\_null) onda ce vratiti 4.

COUNT(\*) broji samo redove dok ce COUNT(qty) brojati kolone sa vrednostima.

Prebrojavanje unikatnih vrednosti po COUNT-u:

COUNT(DISTINCT qty) ili npr SUM(DISTINCT qty)

#### ■ Funkcije MAX & MIN traze po grupi

Ukoliko zelis da trazis po redovima najvecu ili najmanju vrednost u koloni onda mozes da koristis funkcije LEAST & GREATEST

## HAVING Clause

Ovo je group filter. Jedino grupe u kojima se HAVING equateuje sa "true" ce biti prikazane/vracene. False/unknown je discarded.

HAVING clause se procesira tek nakon sto su redovi grupisani pa mozes i koristiti aggregate fn's u filteru.

Npr grupe sa samo preko 5 porucenih artikala:

HAVING COUNT(qty) > 5

ili grupe koje imaju vise od jednog reda u sebi

HAVING COUNT(\*) > 1

## SELECT Clause

Specificiras atribute/kolone koje zelis da se vrate u result tabeli query-a.

Kad pises SELECT \* FROM <tabela>, taj \* ce selektovati kolone redosledom koje su napisane u CREATE TABLE izrazu. To je cesto losa praksa i uvek treba da specificiras sta zelis da selektujes tj kolone. Ako izmenis taj redosled nekako, onda ce u transacionom programu taj \* znaciti drugu stvar i mozes dobiti greske.

Curiously, you are not allowed to refer to column aliases created in the *SELECT* clause in other expressions within the same *SELECT* clause. That's the case even if the expression that tries to use the alias appears to the right of the expression that created it. For example, the following attempt is invalid:

```
SELECT orderid,  
    YEAR(orderdate) AS orderyear,  
    orderyear + 1 AS nextyear  
FROM Sales.Orders;
```

I'll explain the reason for this restriction later in this chapter, in the section "All-at-once operations." As explained earlier in this section, one of the ways around this problem is to repeat the expression:

```
SELECT orderid,  
    YEAR(orderdate) AS orderyear,  
    YEAR(orderdate) + 1 AS nextyear  
FROM Sales.Orders;
```

## ORDER BY Clause

Sluzi da sortira redove rezultata upita. Koristi se za output purposes.

Logicki, izvrsava se posle *SELECT* statementa. Znaci *SELECT* je predposlednji, a *ORDERBY* poslednji.

Posto se redosled clanove ne moze garantovati u tabelama kad se vraca rezultat, a ti zelis garanciju onda moras koristiti *ORDER BY*. Medjutim, ordered redovi se nazivaju "cursori". Sad kako je redosled zagarantovan, tako je i to drugi tip rezultata u pitanju koji ne moze da se "qualify" kao tabela.

U *ORDER BY* mozes da se pozivas na aliase iz *SELECT* statementa. To cak ni u samom *SELECT* statementu ne mozes. Ni u jednom drugom statementu jer jelte, redosled operacija. Ovo ne moze:

```
SELECT name as xname, UNIQUE xname
```

```
FROM tabelica
```

```
WHERE xname LIKE '%kita%'
```

ASC – ascending order by(default)

DESC – descending order by

U *ORDER BY* mozes specificirate i kolone koje nisu specificirane u *SELECT* statementu

```
SELECT name
```

```
FROM users
```

```
ORDER BY age
```

However, when the *DISTINCT* clause is specified, you are restricted in the *ORDER BY* list only to elements that appear in the *SELECT* list.

## The TOP and OFFSET-FETCH filters

Ovo su x2 filtera koji se ne baziraju na predicateu poput WHERE/HAVING nego se baziraju na osnovu broja redova i redosleda(rows & ordering)

### TOP filter

Sluzi da ogranicis broj ili procenat redova koje tvoj query vraca. Oslanja se dakle na sam broj/procenat redova koje vraca i redosled(ordering).

**SELECT TOP(5) orderid FROM Sales.Orders ORDERBY orderdate DESC**

TOP Filter je handled posle distinct. To znaci da ako DISTINCT je specificiran u SELECT clause, top filter je evaluated nakon sto su duplicate rows "razreseni"(removed)

Ukoliko koristis "PERCENT" keyword, onda mozes da uzmes npr top 1% redova(will be rounded up)

**SELECT TOP(1) PERCENT orderid FROM Sales.Orders ORDER BY orderdate DESC**

Npr ako imas 830 redova, rezultat ce biti 9 redova jer 1% rounded up nad 830 je 9.

### The OFFSET-FETCH filter

Problem TOP filtera je sto nije standard i nema "skip" opciju. Enter offset fetch(korisno za paging). OFFSET-FETCH po SQL standardu je considered kao ekstenzija ORDER BY clause-a.

OFFSET clause – koliko redova zelis da skippujes

FETCH clause – koliko redova zelis da filtriras nakon skipa

```
SELECT ordereid, orderdate  
FROM Sales.Orders  
ORDER BY orderdate, orderid  
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY
```

Orderuje prvo po orderdate pa orderid, skippuje 50 redova i uzima sledecih 25.

Uslov za OFFSET-FETCH je da se koristi ORDER BY clause

Takodje FETCH ne mozes da koristis bez OFFSET :)

OFFSET bez fetcha je allowed, a ako zelis da uzmes samo "X" redova bez skippovanja onda:

```
OFFSET 0 ROWS FETCH NEXT 25 ROWS ONLY
```

Imas i "sinonima", znaci umesto "NEXT" mozes da kazes "FIRST"  
Umesto "ROWS" mozes da kazes "ROW". Ista funkcionalnost, samo drugaciji izraz. Cilj je da bude more english like.

## Window functions – quick look

Window function je funkcija koja za svaki red u underlying query, operates on a window(set) of rows koji deriveuju iz underlying query resulta i computuje skalarni(jedan) result value.

Window of rows je definisan sa OVER clauseom. Window funkcije su profound(raznolike).

Bilo bi prerano ulaziti u detalje sada, evo nekih.

ROW\_NUMBER window function.

Znaci, window funkcija radi na skupu redova koji su exposeovani kroz "OVER" clause. Za svaki row, OVER clause exposeuje funkciji podskup redova iz glavnog query resula.

OVER clause moze da restrictuje redove u window-u koriscenjem optional window partition clause(PARTITION BY). Takodje mozes definisati ordering za calculation koriscenjem window order clause(ORDER BY) koji se ne vezuje za query-ev ORDER BY.

Primer:

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                          q           ORDER BY val) as rounum
  FROM Sales.OrderValues
 ORDER BY custid, val

SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                          ORDER BY val) AS rounum
  FROM Sales.OrderValues
 ORDER BY custid, val;
```

This query generates the following output, shown here in abbreviated form:

orderid	custid	val	rounum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			

ROW\_NUMBER ce za svaku particiju generisati unique sekvencialno incrementing integere koji predstavljaju redove u rezultatu po particiji. Dakle particija je kao neki interni group by i tu dobijas “podskup”.

Kreira se dakle separate particija za svaki distinct customer id value, pa s toga row number su unique za svakog customera. OVER clause takođe definise ordering u window na osnovu val atributa, pa ce sequential row numbers biti incremented sa particijom na osnovu vrednosti atributa.

NOTE:

Expressioni u SELECT se evaluiraju PRE distinct clause-a(ako postoji distinct). Iz tog razloga ce se prvo WINDOW pa onda DISTINCT na kraju.

## Predicates & Operators

U sustini “Predicate” je izraz koji vraca TRUE/FALSE/UNKNOWN. Moze se redjati u kombinaciji sa NOT, AND ili OR operatorima.

Ovde prica o IN, BETWEEN i LIKE predicateima.

TSQL supports operatore: =, >, <, >=, <=, <>, !=, !=, !>, !<. Zadnja x3 nisu standardna. != je zapravo <>

Recommendation je ne koristiti nonstandard opratore.

Od aritmetike: +, -, \*, /, %

Mozes mesati data types i T-SQL ce operand sa vecom “data type precedence” uzeti i staviti kao data type skalarног izraza.

Dakle x2 integera 5/2 racaju 2, a ne 2.5.

Zato moras uraditi cast:

CAST(col1 AS NUMERIC(12,2)) / CAST(col2 AS NUMERIC(12,2))

NUMERIC(12,2) bi bio precision od 12 i 2 mesta posle tacke.

Ako uradis neki INT/NUMERIC tj 5/2.0 onda ces dobiti 2.5 kao rezultat

## OPERATOR PRECEDENCE:

1. ( ) (Parentheses)
2. \* (Multiplication), / (Division), % (Modulo)
3. + (Positive), – (Negative), + (Addition), + (Concatenation), – (Subtraction)
4. =, >, <, >=, <=, <>, !=, !=, !< (Comparison operators)
5. NOT
6. AND
7. BETWEEN, IN, LIKE, OR
8. = (Assignment)

## CASE Expressions

CASE expression je scalar expression koji braca vrednost na osnovu “conditional logic”.

CASE nije expression niti statement, nece kontrolisati flow koda. Znaci samo vraca vrednost. Kako je scalar expression onda se moze koristiti gde kog su i skalarni izrazi dozvoljeni: SELECT, WHERE, HAVING, ORDER BY i CHECK constraints

Postoje x2 tipa CASE expressions:

- **Simple**
- **Searched**

**Simple** je za poredjenje vrednosti/izraza sa listom mogucih vrednost i vracanje vrednosti na prvom matchu. Ako nema, onda se vraca vrednost u ELSE clause. Ukoliko nema ELSE, onda se defaultuje na ELSE NULL.

```
SELECT supplierid, COUNT(*) AS numproducts,
CASE COUNT(*) % 2
    WHEN 0 THEN 'Even'
    WHEN 1 THEN 'Odd'
    ELSE 'Unknown'
END AS coutparity
FROM Production.Products GROUP BY supplierid
```

**Searched** case expression dozvoljava definisanje predikata u WHEN clause umesto da budes ogranicen samo na equality comparisons.

CASE

```
WHEN val < 1000.00 THEN 'Kita'  
WHEN val >= 1000.00 THEN 'Velika kita'  
ELSE 'Unknown'  
END AS valuecategory
```

## NULLS

Dakle predicate logic pored TRUE i FALSE moze da evaluateuje i "UNKNOWN". Na primer u slucaju NULL-a ce predicate salary > 0 evaluateovani "UNKNOWN" kada je salary null.

Ukoliko je iznad 0 onda TRUE, a ako je ispod 0 onda FALSE.

Razliciti statementi razlicito na ovo gledaju jer UNKNOWN nije false.

Filteri poput WHERE i HAVING imaju "accepts TRUE" treatment. To znaci da u WHERE samo oni koji u predicate vracaju TRUE ce biti uzeti u obzir.

CHECK constraint ima statement "rejects FALSE" sto bi znacilo da bi i UNKNOWN i TRUE bio prihvacen.

Obrati paznju na to.

Ukoliko uradis negaciju na unknown, ti i dalje dobijas "unknown".

NOT (salary > 0) -> NOT UNKNOWN == UNKNOWN.

NULL = NULL ce takodje evaluateovati na UNKNOWN.

NULL predstavlja odsustvo vrednosti pa iz tog razloga imas operatore:

IS NULL i IS NOT NULL, which you should use instead of = NULL and <> NULL.

SQL Server 2022 unosi promenu koja se naziva **IS [NOT] DISTINCT FROM** clause koja tretira NULL's kao values i za rezultat dobijas iskljucivo true/false bez unknown.

<operand1> IS NOT DISTINCT FROM <operand2> - ovo se evaluira kao (=) sto evaluira u TRUE kada se porede dve null vrednosti. Ako je NULL sa non NULL onda je FALSE.

<operand1> IS DISTINCT FROM <operand2> - ovo se evaluira kao (<>) sto evaluira u FALSE kada se porede dve NULL vrednosti. Ako je Non null i Null onda je TRUE.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region IS NOT DISTINCT FROM N'WA';
```

je ideticno sa ==

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = N'WA';
```

Koriscenje two valued modela ima smisla kada poredis dve kolone ili kolone sa konstantama ili npr u stored procedurama/trigerima.

Ranije si morao pisati ovako nesto:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = @region
OR (region IS NULL AND @region IS NULL);
```

Ukoliko je @region NULL, onda je ovo jedini nacin da dobijes TRUE na NULL = NULL ako je i region column u trenutnom row-u na NULL.

Da bi to izbegli mozemo napisati:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region IS NOT DISTINCT FROM @region
```

Kada su NULL IS NOT DISTINCT FROM NULL, onda je rezultat TRUE

Isto vazi i za <> i IS DISTINCT FROM; tj razlicistost.

‘Kita’ IS DISTINCT FROM NULL ti vrati “true”

Kako se drugi elementi SQL-a ponasaju prema NULLovima?

**GROUP BY** – grupise sve NULL-ove zajedno

**ORDER BY** – sortira sve nullove zajedno; u T-SQL u ascending orderu prvo idu null, onda vrednosti

**UNIQUE** – po standardnom SQL-u UNIQUE enforceuje unique na non null values. T-SQL gleda na NULL kao vrednost pa mozes samo x1 null imati.

## GREATEST and LEAST functions

Od sql server 2022 mozes stavljati najvece i najmanje vrednosti redova:

However, sometimes you need to apply maximum and minimum calculations across columns, or across a set of expressions, **PER ROW**

```
SELECT orderid, requireddate, shippeddate,  
       GREATEST(requireddate, shippeddate) AS latestdate,  
       LEAST(requireddate, shippeddate) AS earliestdate  
FROM Sales.Orders  
WHERE custid = 8;
```

orderid	requireddate	shippeddate	latestdate	earliestdate
10326	2020-11-07	2020-10-14	2020-11-07	2020-10-14
10801	2022-01-26	2021-12-31	2022-01-26	2021-12-31
10970	2022-04-07	2022-04-24	2022-04-24	2022-04-07

**NULL inputs are ignored, but if all inputs are NULL, the result is NULL.**

Ovo znaci radi poredjenje u samom ROW-u:

**GREATEST(val1, expression\_2, val3) i onda vraca najvecu od tih.**

Ranije:

These functions support any number of input arguments between 1 and 254. *NULL* inputs are ignored, but if all inputs are *NULL*, the result is *NULL*.

In earlier versions of SQL Server, if you had only two input arguments, you could handle such needs with CASE expressions, like so:

```
SELECT orderid, requireddate, shippeddate,  
CASE  
    WHEN requireddate > shippeddate OR shippeddate IS NULL THEN requireddate  
    ELSE shippeddate  
END AS latestdate,  
CASE  
    WHEN requireddate < shippeddate OR shippeddate IS NULL THEN requireddate  
    ELSE shippeddate  
END AS earliestdate  
FROM Sales.Orders  
WHERE custid = 8;
```

## All at once operations

Ovo znači da T-SQL podržava feature kada svi izrazi koji se pojave u istoj fazi "logickog query processinga" su evaluirani u isto vreme.

Svi izrazi koji se pojavljuju u istoj fazi su tretirani kao "set" i kao ranije pomenuto – set/skup nema redosleda.

This concept explains why, for example, you cannot refer to column aliases assigned in the SELECT clause within the same SELECT clause

Drugi fuckup je short-circuit expressiona sto T-SQL podržava, ali zbog all at once operations, možda ti se evaluation ne dogodi kako mislis da treba.

```
SELECT col1, col2
FROM dbo.T1
WHERE col1 <> 0 AND col2/col1 > 2;
```

Dakle, ako je  $\text{col1} \neq 0$  FALSE, onda se ovaj drugi deo neće uzimati u obzir. Ali možda se prvo evaluira  $\text{col2}/\text{col1} > 2$  i dobijes error ako je  $\text{col1} = 0$ . Nema divide by zero u matematici :)

SQL Server cesto uzima u obzir estimacije po pitanju kompleksnosti izraza i na osnovu toga može neki izraz ranije, a neki kasnije da evaluira.

U ovom gore slučaju kako bi smo izbegli failure možemo koristiti **CASE** expression sa **WHEN** jer je tu redosled zagarantovan.

```
SELECT col1, col2
FROM dbo.T1
WHERE
CASE
WHEN col1 = 0 THEN 'no'
WHEN col1 <> 0 THEN 'yes'
ELSE 'no'
END 'yes'
```

Zbog kompleksnosti tog workarounuda imamo bolju opciju

```
SELECT col1, col2 FROM dbo.T1
WHERE (col1 > 0 AND col2 > 2*col1) OR (col1 < 0 AND col2 < 2*col1);
```

Think about this :)

## Working with CHARACTER data

Postoje x2 tipa character data:

- regular
- n kind

Regular su npr CHAR ili VARCHAR

n kind su NCHAR i NVARCHAR

N je “national”, to je kao znacenje tog prefixa.

Razlika je da N kind podrzavaju UTF-16 dok regular podrzavaju samo UTF-8.

Kada navodis velicinu karaktera npr VARCHAR(10) ti kazes da tu moze da se smesti 10B karaktera. U teoriji bi to znacilo 10 karaktera, ali u praksi neki karakter moze da zauzima i 2B ili vise prostora.

Kada referujes na regular type literal onda koristis single quotes ‘this is a literal’

Kada referujes na n kind type literal onda koristis “N” prefix pre quotes N’this is a literal’

Character data types koji nemaju “VAR” u sebi, oni podrzavaju samo onoliko prostora koliko im kazes. Fixed size su. Iz tog razloga su bolji za writefocused systems, ali nisu pogodni toliko za read focused systems jer storage consumption nije toliko optimalno.

NVARCHAR(25) ce mu staviti 25B prostora + 2B za offset data ali u praksi ce user data odluciti o tome koliko prostora ce se tu zauzeti jer je varijabilno jelte. Because storage consumption for these data types is less than that for fixedlength types, read operations are faster.

Ako stavis velicinu na (MAX) to je onda 2GB prostora. Any value with a size up to a certain threshold (8,000 bytes by default) can be stored inline in the row. Any value with a size above the threshold is stored external to the row as a large object (LOB)

## Collation

Collation je svojstvo charater data koje enkapsulira nekoliko aspekata:

- Sort order
- Case sensitivity
- ...jos :)

Da dobijes skup supported collationa i njihovih opisa onda pozivas fn\_helpcollations

```
SELECT name, description
FROM sys.fn_helpcollations();
```

- **Latin1\_General** Code page 1252 is used. (This supports English and German characters, as well as characters used by most Western European countries.)
- **Dictionary sorting** Sorting and comparison of character data are based on dictionary order (A and a < B and b).

## CHAPTER 2 Single-table queries

You can tell that dictionary order is used because that's the default when no other ordering is defined explicitly. More specifically, the element *B/N* doesn't explicitly appear in the collation name. If the element *B/N* appeared, it would mean that the sorting and comparison of character data was based on the binary representation of characters (A < B < a < b).

- **CI** The data is case insensitive (a = A).
- **AS** The data is accent sensitive (à < > ä).

Mozes da podesavas COLLATION izraza(querya) koriscenjem **COLLATE** clause

```
SELECT * FROM Customers  
WHERE lastname COLLATE Latin1_General_CS_AS = N'Davis'
```

I ovo ce collate na case sensitive jer "CS". Ako nemas collate, onda ce se assumeovati default collation.

## Operations and Functions nad characters

**Konkatonacija:** + sign operator, CONCAT, CONCAT\_WS

For **other operations:** SUB STRING, LEFT, RIGHT, LEN, DATALENGTH, CHARINDEX, PATINDEX, REPLACE, TRANSLATE, REPLICATE, STUFF, UPPER, LOWER, RTRIM, LTRIM, TRIM, FORMAT, COMPRESS, DECOMPRESS, STRING\_SPLIT, STRING\_AGG

## COALESCE FN

Standard SQL kaze da konkatonacija sa "NULL" yielduje sa "NULL" kao rezultatom.

region + N' ' + city => NULL ukoliko je i jedna od region/city NULL

Kako bi se NULL tretirao kao empty string onda se koristi **COALESCE function.**

```
SELECT custid, country, region, city,
    country + COALESCE(N'' + region, N'') + N'' + city AS location
FROM Sales.Customers;
```

Coalesce funkcija kad naidje na prvu null, onda samo returnuje empty string.

Potomo imamo CONCAT

## CONCAT FN

Prima se lista charactera i automatski zameni NULL sa empty string

CONCAT('Jure', N' je', NULL, ' derpe') => Jure je derpe

## CONCAT\_WS

Ovo je konkatonacija sa separatorom. Prvi character je separator. Svaki sledeci character ce se razdvojiti tim separatorom

CONCAT('x', 'Jure', N' je', NULL, ' derpe') => Jure x je x derpe; valjda ovako

## SUBSTRING

SUBSTRING(string, start, length)

SELECT SUBSTRING('abcde', 1, 3); => 'abc'

Ne ide od 0 start. Takođe ako je length > od lengtha stringa, onda ce se jednostavno vratiti sve bez da se baca neki error

## LEFT & RIGHT

Substring s leva ili desna

LEFT('string', length) & RIGHT('string', length)

RIGHT('string', 3) => ing;

## LEN & DATALENGTH

**LEN('string')** vraca length stringa

**DATALENGTH('string')** vraca koliko B zauzima string

Another difference between LEN and DATALENGTH is that the former excludes trailing spaces but the latter doesn't.

## CHARINDEX

**CHARINDEX(substring, string[, start\_pos])** vraca poziciju prvog occurrencea substringa unutar stringa.

```
SELECT CHARINDEX(' ', 'Itzik Ben-Gan'); => 6
```

## PATINDEX

**PATINDEX(pattern, string)** vraca prvi occurrence patterna u stringu

Pattern koristi slican izraz poput patterna u LIKE-u

```
SELECT PATINDEX('%[0-9]%', 'abcd123efgh'); => 5; vrati index prvog broja
```

## REPLACE

**REPLACE(string, substr1, substr2)** zamenjuje sve occurrences substring1 sa substring 2

```
SELECT REPLACE('1-a 2-b', '-', ':'); This code returns the output '1:a :b'.
```

Sa REPLACE mozes prebrojavati i broj occurancesa nekog karaktera. Na primer karakter koji trazis(zelis da prebrojas) mozes da zamenis empty stringom. Izracunas LEN rezultata i oduzmes ga od LEN izvornog stringa.

```
SELECT (LEN(src_strng) – LEN(REPLACE(src_strng, ‘a’)))
```

## REPLICATE

Replicate function replicates a string “x” number of times

```
REPLICATE(string, n)
```

```
REPLICATE(‘abc’,3) => abcabcabc
```

Na primer combo sa RIGHT funkcijom

**SELECT supplierid,**

```
RIGHT(REPLICATE(‘0’, 9) + CAST(supplierid AS VARCHAR(10)), 10) AS strsupplierid  
FROM Production.Suppliers;
```

supplierid	strsupplierid
29	0000000029
28	0000000028
4	0000000004

## STUFF

Ova funkcija sluzi da se substring unutar stringa zameni drugim substringom.

STUFF(string, pos, delete\_length, insert\_string)

Na primer:

STUFF('xyz', 2, 1, 'kita') => xkitaz

## UPPER & LOWER

Za uppercase i lowercase

## LTRIM, RTRIM & TRIM

Dozvoljava ti da removeujes leading/trailing ili oba leading & trailing characters.

To znaci removeovanje spaceova :)

LTRIM(string)

## FORMAT

Sluzi za formatiranje inputa na osnovu msft .net format stringa i culture-a.

**FORMAT(input , format\_string, culture)**

FORMAT(10, 'd10') => 0000000010

FORMAT je skupa funkcija i ne bi je trebalo koristiti

## COMPRESS & DECOMPRESS

Kompresuju i dekompresuju po GZIP algoritmu

COMPRESS(string)

DECOMPRESS(string)

Prihvata character ili binary string kao input i vraca VARBINARY(MAX) typed value

Recimo imas stored procedure i zelis da storeujes CV-eve kandidata u bazi koji su kompresovani

INSERT INTO dbo.EmployeeCVs( empid, cv ) VALUES( @empid, COMPRESS(@cv) );

Ovaj @cv input je NVARCHAR(MAX)

Ovaj @empid je input parametar za employee

Kolona "cv" u dbo.EmployeeCVs ce biti tipa VARBINARY(MAX)

**DECOMPRESS** prihvata binary string kao input i vraca VARBINARY(MAX) kao output. Da bi dobio originalnu vrednost nazad, moraces castovati taj varbinary u ono sto ocekujes:

```
SELECT CAST(DECOMPRESS(COMPRESS(N'This is my cv. Imagine it was much longer.')) AS NVARCHAR(MAX));
```

Pa na primer za onu kolonu gore bi taj decompress izgledao ovako:

```
SELECT empid,
      CAST(DECOMPRESS(cv) AS NVARCHAR(MAX)) AS cv
   FROM dbo.EmployeeCVs;
```

## STRING\_SPLIT

Separateuje string u listu individualnih vrednosti

```
SELECT value FROM STRING_SPLIT(string, separator[, enable_ordinal]);
```

Za razliku od svih dosadasnjih funkcija, ovo je prva table function.

```
SELECT CAST(value AS INT) AS myvalue
FROM STRING_SPLIT('10248,10249,10250', ',')
```

Ordinal flag moze ako imas mssql 2022 or later.

```
SELECT CAST(value AS INT) AS myvalue, ordinal
FROM STRING_SPLIT('10248,10249,10250', ',', 1)
```

Ovo je potom rezultat

myvalue	ordinal
10248	1
10249	2
10250	3

## STRING\_AGG

Aggregate funkcija agregira listu stringova u jedan

```
STRING_AGG(input , separator) [WITHIN GROUP(order_specification) ]
```

U sustini u toj target grupi funkcija konkatonira vrednosti input argumenta i razdvaja ih na osnovu separatora.

Da bi radio garanciju nad redosledom vrednosti, onda se koristi WITHIN GROUP

Naredni query agregira order ID-eve svakog customera rasporedjenih po najskorijem i razdvojeni zarezom

```
SELECT custid,
       STRING_AGG(CAST(orderid AS VARCHAR(10)), ',')
           WITHIN GROUP(ORDER BY orderdate DESC, orderid DESC) AS custorders
    FROM Sales.Orders
   GROUP BY custid
```

custid	custorders
1	11011,10952,10835,10702,10692,10643
2	10926,10759,10625,10308
3	10856,10682,10677,10573,10535,10507,10365
...	

(89 rows affected)

NOTE:

Output STRING\_AGG je **nvarchar** osim ukoliko input nije **varchar**. U tom slucaju je i output **varchar**.

## LIKE Predicate

Koristi se da se isproverava da li character string matchuje specified pattern.

Characters:

- % => The percent(any size) wildcard
- \_ => The underscore(single character) wildcard
- [ABZ] => The [<list of characters>] wildcard
- [A-Z] => The [<character-character>] wildcard
- [A^B-Z] => The [^<character list or range>] wildcard

## Working with date and time data

Ovo ludilo podrzava x6 date and time formata.

DATETIME & SMALLDATETIME => Kao legacy formati  
DATE, TIME, DATETIME2 & DATETIMEOFFSET

Table 2-1 lists details about date and time data types, including storage requirements, supported date range, precision, and recommended entry format.

**TABLE 2-1** Date and time data types

Data type	Storage (bytes)	Date range	Precision	Recommended entry format and example
<i>DATETIME</i>	8	January 1, 1753, through December 31, 9999	3 1/3 milliseconds	'YYYYMMDD hh:mm:ss.nnn' '20220212 12:30:15.123'
<i>SMALLDATETIME</i>	4	January 1, 1900, through June 6, 2079	1 minute	'YYYYMMDD hh:mm' '20220212 12:30'
<i>DATE</i>	3	January 1, 0001, through December 31, 9999	1 day	'YYYY-MM-DD' '2022-02-12'
<i>TIME</i>	3 to 5	N/A	100 nanoseconds	'hh:mm:ss.nnnnnnnn' '12:30:15.1234567'
<i>DATETIME2</i>	6 to 8	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnnn' '2022-02-12 12:30:15.1234567'
<i>DATETIMEOFFSET</i>	8 to 10	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnnn [+/-]hh:mm' '2022-02-12 12:30:15.1234567 +02:00'

Storage poslednja x3 zavisi od preciznosti koju odaberes

## Literals

Kada treba da definises konstantu za date & time trebas da razmislis o nekoliko stvari. Naime T-SQL nema bas prirodan nacin za definisanje Date & Time konstanti pa moras definisati konstantu jednog tipa i potom je konvertovati u date & time.

Postoje i implicit konverzije, ali je kompletan fuckup to razumeti.

**SELECT orderid, custid, empid, orderdate**

**FROM Sales.Orders**

**WHERE orderdate = '20220212';**

⇒ Naime ovaj gore query je validan :) Ovaj dole ce biti uz cast:

**SELECT orderid, custid, empid, orderdate**

**FROM Sales.Orders**

**WHERE orderdate = CAST('20220212' AS DATE);**

**TABLE 2-2** Language-neutral date and time data type formats

Data type	Recommended entry format	Example
<i>DATETIME</i>	'YYYYMMDD hh:mm:ss.nnn' 'YYYY-MM-DDThh:mm:ss.nnn' 'YYYYMMDD'	'20220212 12:30:15.123' '2022-02-12T12:30:15.123' '20220212'
<i>SMALLDATETIME</i>	'YYYYMMDD hh:mm' 'YYYY-MM-DDThh:mm' 'YYYYMMDD'	'20220212 12:30' '2022-02-12T12:30' '20220212'
<i>DATE</i>	'YYYYMMDD' 'YYYY-MM-DD'	'20220212' '2022-02-12'

Data type	Recommended entry format	Example
DATETIME2	'YYYYMMDD hh:mm:ss.nnnnnnnn'	'20220212 12:30:15.1234567'
	'YYYY-MM-DD hh:mm:ss.nnnnnnnn'	'2022-02-12 12:30:15.1234567'
	'YYYY-MM-DDThh:mm:ss.nnnnnnnn'	'2022-02-12T12:30:15.1234567'
	'YYYYMMDD'	'20220212'
	'YYYY-MM-DD'	'2022-02-12'
DATETIMEOFFSET	'YYYYMMDD hh:mm:ss.nnnnnnnn [+ -]hh:mm'	'20220212 12:30:15.1234567 +02:00'
	'YYYY-MM-DD hh:mm:ss.nnnnnnnn [+ -]hh:mm'	'2022-02-12 12:30:15.1234567 +02:00'
	'YYYYMMDD'	'20220212'
	'YYYY-MM-DD'	'2022-02-12'
TIME	'hh:mm:ss.nnnnnnnn'	'12:30:15.1234567'

Nmg vise o ovome :)

## Filtering Date Ranges

Ako moras da filtriras po mesecu ili po godini, ima smisla koristiti YEAR i MONTH funkcije

```
SELECT * FROM x WHERE YEAR(orderdate) = 2021
```

```
SELECT * FROM x WHERE MONTH(orderdate) = 2
```

ALI, vidis dragi moj naivni citaoce. Ovo sjebava indeksiranje :)

Ne znam kako ni zasto, ali u sustini ne moze se efikasno iskoristiti pa bi se trebalo pridrzavati obicnih logickih operacija :)

**Month varijanta:** =>-----

```
SELECT orderid, custid, empid, orderdate
```

```
FROM Sales.Orders
```

```
WHERE orderdate >= '20220201' AND orderdate < '20220301';
```

**YEAR varijanta:** =>-----

```
SELECT orderid, custid, empid, orderdate
```

```
FROM Sales.Orders
```

```
WHERE orderdate >= '20210101' AND orderdate < '20220101';
```

## Date and time functions

GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, SYSDATETIMEOFFSET, CAST, CONVERT, PARSE, SWITCHOFFSET, TODATETIMEOFFSET, AT TIME ZONE, DATEADD, DATEDIFF and DATEDIFF\_BIG, DATEPART, YEAR, MONTH, DAY, DATENAME, DATETRUNC, DATE\_BUCKET, ISDATE, various FROMPARTS functions, EOMONTH, and GENERATE\_SERIES.

## GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, SYSDATETIMEOFFSET

Function	Return type	Description
<code>GETDATE</code>	<code>DATETIME</code>	Current date and time
<code>CURRENT_TIMESTAMP</code>	<code>DATETIME</code>	Same as <code>GETDATE</code> but SQL-compliant
<code>GETUTCDATE</code>	<code>DATETIME</code>	Current date and time in UTC
<code>SYSDATETIME</code>	<code>DATETIME2</code>	Current date and time
<code>SYSUTCDATETIME</code>	<code>DATETIME2</code>	Current date and time in UTC
<code>SYSDATETIMEOFFSET</code>	<code>DATETIMEOFFSET</code>	Current date and time, including the offset from UTC

## CAST, CONVERT, PARSE & their TRY\_Counterparts

Sluze za konvertovanje input value-a u neki drugi target type. TRY\_verzija ovih funkcija ako konverzija failuje ce vratiti NULL umesto da failuje.

`CAST(value as target_data_type) => SELECT CAST(SYSDATETIME() AS DATE)`  
`CONVERT(target_data_type, value, [style_number])`  
`PARSE(value as target_data_type [USING culture])`

`TRY_CAST`, `TRY_CONVERT`, `TRY_PARSE` isto sve.

Treci opcioni argument je za definisanje npr stila konverzije tj npr kad iz char -> datetime pa hoces da definises u kom formatu si pisao taj datetime DD/MM/YYYY umesto YYYY-MM-DD na primer.

## SWITCHOFFSET Function

Podesava vrednost DATETIMEOFFSET-a u specificirani target offset “from UTC”.

`SWITCHOFFSET(datetimeoffset_value, UTC_offset)`

Umm `SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');`

So if the current system datetimeoffset value is February 12, 2022 10:00:00.0000000 – 08:00, this code returns the value February 12, 2022 13:00:00.0000000 –05:00

`SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+00:00');`

Assuming the aforementioned current datetimeoffset value, this code returns the value February 12, 2022 18:00:00.0000000 +00:00.

## TODATETIMEOFFSET Function

Kreira DATETIMEOFFSET typed value iz lokalnog date and time value i offset-a iz UTC

### **TODATETIMEOFFSET(local\_date\_and\_time\_value, UTC\_offset)**

You will typically use this function when migrating non-offset-aware data to offset-aware data.

I dalje mislim da ne kontam ovo.

## The AT TIME ZONE Function

Uzima date and time value i konvertuje ga u **datetimeoffset** type za taj specified time zone.

### **date\_time\_value AT TIME ZONE time\_zone**

Supported inputs: DATETIME, SMALLDATETIME, DATETIME2, and DATETIMEOFFSET

--- ima jos dosta o ovome, nadam se da mi nikada nece trebati.

## The DATEADD function

### **DATEADD(part, n, dt\_val)**

Dodaje "jedinice"(n) na specificirani date part u date time value

Validne vrednosti za part su:

*year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, and nanosecond*

DATEADD(year, 1, '20220212') => 2023-02-12 00:00:00.000

## DATEDIFF & DATEDIFF\_BIG Functions

Vracaju razliku izmedju x2 date and times na osnovu specified date part-a.

DATEDIFF vraca rezultat u INT

DATEDIFF\_BIG vraca rezultat u BIGINT type-u

DATEDIFF(date\_part, date\_1, date\_2) :INT

DATEDIFF\_BIG(date\_part, date\_1, date\_2) :BIGINT

**SELECT DATEDIFF(day, '20210212', '20220212');** => 365

Vrati ti razliku u danima, jelte.

## **DATEPART Function**

Vraca integer koji predstavlja specified date\_part

**DATEPART(date\_part, date) :INT**

DATEPART(month, '20220221) => 2

Mozes koristiti i *day*, *weekday*, *dayofyear* kao part parametre.

## **YEAR, MONTH & DAY Functions**

Poticu iz DATEPART funkcije. Isti k samo sto jelte lakse za izrazavanje

**YEAR(date) == DATEPART(year, date)**

## **DATENAME Function**

Vraca character string koji predstavlja “part” tog date\_value-a.

**DATENAME(date\_value, part);**

DATENAME('20220201', month) => 'February'

Ukoliko ti je session language podesen na english dobices February. Ako ti je na italijanskom, onda dobijer “Februararoaroiujasfi” iliti sta god da je na italijanskem februar.

Ako requested part nema naziv(npr godina), onda samo dobijes broj koji je zapravo character :)

DATENAMT('20220201', year) => '2022'

## **DATETRUNC Function**

DATETRUNC truncateuje tj. flooruje input date and time value do “pocetka specificiranog dela”

Ovo je predstavljeno u SQL SERVER 2022

### **DATETRUNC(part, date\_value)**

Ako je date\_value tipa DATETIME, onda ce i output biti DATETIME. Ako je date\_value character string, onda ce output biti DATETIME2(7)

Valid part vrednosti: *year*, *quarter*, *month*, *dayofyear*, *day*, *week*, *iso\_week*, *hour*, *minute*, *second*, *millisecond*, and *microsecond*

DATETRUNC(month, '20120224') => 2012-02-01 00:00:00.000

## ISDATE Function

Uzima character input i vraca 1/0(bool) ukoliko je character valid date ili nije

ISDATE('2022123493') => 0

ISDATE('20220221') => 1

## FROMPARTS Function

FROMPARTS funkcija prihvata integer input koji predstavljaju delove date & time vrednosti i konstruisu vrednost “requested tipa” iz tih delova

DATEFROMPARTS(year, month, day)

DATETIME2FROMPARTS(year, month, day, hour, minute, seconds, fractions, precision)

DATETIMEFROMPARTS(year, month, day, hour, minute, seconds, milliseconds)

DATETIMEOFFSETFROMPARTS(year, month, day, hour, minute, seconds, fractions, hour\_offset, minute\_offset, precision)

SMALLDATETIMEFROMPARTS(year, month, day, hour, minute)

TIMEFROMPARTS(hour, minute, seconds, fractions, precision)

**SELECT**

```
DATEFROMPARTS(2022, 02, 12),
DATETIME2FROMPARTS(2022, 02, 12, 13, 30, 5, 1, 7),
DATETIMEFROMPARTS(2022, 02, 12, 13, 30, 5, 997),
DATETIMEOFFSETFROMPARTS(2022, 02, 12, 13, 30, 5, 1, -8, 0, 7),
SMALLDATETIMEFROMPARTS(2022, 02, 12, 13, 30),
TIMEFROMPARTS(13, 30, 5, 1, 7);
```

## EOMONTH Function

Prihvata datum(date and time value) i vraca u DATE formatu respective end of month date

EOMONTH(input [, months\_to\_add])

**SELECT EOMONTH(SYSDATETIME())**

## The GENERATE\_SERIES function

Ovo je table funkcija koja vraca sekvencu brojeva u zahtevanom range-u.

Introduced in MSSQL 2022

```
GENERATE_SERIES(start_value, stop_value [, step_value])
```

Specificiras start i stop values i optionally step. Mozes -1 za step kako bi dobio decresing range, to probaj kad budes zapisivao u svesku.

```
SELECT value  
FROM GENERATE_SERIES(1, 10)
```

```
value  
-----  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Mozes se malo igrati pa konvertovati u dane npr:

```
DECLARE @startdate AS DATE = '20220101', @enddate AS DATE = '20221231';  
  
SELECT DATEADD(day, value, @startdate) AS dt  
FROM GENERATE_SERIES( 0, DATEDIFF(day, @startdate, @enddate) ) AS N;
```

```
dt  
-----  
2022-01-01  
2022-01-02  
2022-01-03  
2022-01-04  
2022-01-05  
...  
2022-12-27  
2022-12-28  
2022-12-29  
2022-12-30  
2022-12-31
```

(365 rows affected)

## Querying metadata

SQL Server ti nudi alate za queryovanje metadata objekata poput informacija u vezi tabela u bezi i kolona u tabeli.

To su catalog views, information schema views i system stored procedures & functions.

## Catalog views

Catalog views nude detaljisane informacije o objektima u bazi podataka, uključujući i SQL server specific data.

### SYS.TABLES VIEW

Npr zelis da izlistas tabele u bezi podataka uključujući i njihove schema.

To mozes kroz queryovanje **sys.tables** view-a

```
SELECT SCHEMA_NAME(schema_id) as table_schema_name  
      , name as table_name  
  FROM sys.tables;
```

SCHEMA\_NAME konvertuje schema ID integer u njegov naziv.

table_schema_name	table_name
HR	Employees
Production	Suppliers
Production	Categories
Production	Products
Sales	Customers
Sales	Shippers
Sales	Orders
Sales	OrderDetails
Stats	Tests
Stats	Scores
dbo	Nums

### SYS.COLUMNS Table

Da dobijes informacije o kolonama u tabeli.

```
SELECT name AS column_name  
      , TYPE_NAME(system_type_id) AS column_type  
      , max_length  
      , collation_name  
      , is_nullable  
  FROM sys.columns WHERE object_id = OBJECT_ID(N'Sales.Orders')
```

column_name	column_type	max_length	collation_name	is_nullable
orderid	int	4	NULL	0
custid	int	4	NULL	1
empid	int	4	NULL	0
orderdate	date	3	NULL	0
requireddate	date	3	NULL	0
shippeddate	date	3	NULL	1

## Information schema views

SQL Server supports skup viewova koji se nalaze u schema "INFORMATION\_SCHEMA"

Npr:

**INFORMATION\_SCHEMA.TABLES** view pokazuje base tabele u bazi zajedno sa njihovim schema names

```
SELECT TABLE_SCHEMA, TABLE_NAME  
FROM INFORMATION_SCHEMA.TABLES  
WHERE TABLE_TYPE = N'BASE TABLE';
```

**INFORMATION\_SCHEMA.COLUMNS** nudi vecinu dostupnih informacija o kolonama:

```
SELECT COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH,  
COLLATION_NAME, IS_NULLABLE  
FROM INFORMATION_SCHEMA.COLUMNS  
WHERE TABLE_SCHEMA = N'Sales' AND TABLE_NAME = N'Orders';
```

## System stored procedures & functions

Internally one prave upite nad system catalog i vracaju malo razumnije metatata informacije.

Primeri:

**EXEC sp\_tables** -> vraca listu objekata poput tabela i viewova koji mogu biti queryovani u trenutnoj bazi podataka

**EXEC sys.sp\_help @objname = 'Sales.Orders'** -> vraca multiple result set sa osnovnim informacijama o objektu i info o kolonama, indexima, constraints & vise za neku tabelu.

**EXEC sys.sp\_columns**

```
@table_name = 'Orders'  
@table_owner = 'Sales' -> info o kolonama u objektu
```

**EXEC sys.sp\_helpconstraint @objname = N'Sales.Orders'** -> info o constraintovima u objektu.

Potom imamo SERVERPROPERTY funkciju koja vraca requested property trenutne server instance

**SELECT SERVERPROPERTY('Collation');** -> Vraca collation trenutne instance

**SELECT DATABASEPROPERTYEX(N'TSQLV6', 'Collation')** -> Vraca property neke specificirane baze

**SELECT OBJECTPROPERTY(OBJECT\_ID(N'Sales.Orders'), 'TableHasPrimaryKey')** ->  
Vraca property specificiranog object name-a, npr ovo gore naznacava da li tabela ima primary key

**SELECT COLUMNPROPERTY(OBJECT\_ID(N'Sales.Orders'), N'shipcountry', 'AllowsNull')**  
-> vraca requested property za kolonu. Npr ovaj gore output govori o tome da li je kolona nullable ili nije.

## Exercises Sve do 116. strane

## JOINS

Unutar FROM clause-a table operatori mogu da vrse operacije nad tabelama.

T-SQL supports x4 tipa table operatora:

- Join (standard); ostali su non standard tj extenzije
- Apply
- Pivot
- Unpivot

JOIN vrsi operacije nad x2 input tabele. Postoje cross joins, inner join i outer join.

## CROSS JOIN

Implementira samo x1 logical query processing fazu – a Cartesian Product.

U ovoj fazi radi se nad dveju tabela kao input i daje se Cartesian Product kao rezultat ove dve(Kartezijski proizvod?). U prevodu, svaki red iz jednog inputa je matchovan sa svakim rowom iz drugog. Matematicki: tabela\_1 ima  $m$  redova, a tabela\_2 ima  $n$  redova.

Rezultat je  $m \times n$

Postoje x2 sintakse -> SQL-92(recommended) & SQL-89

### SQL-92 sintaksa:

```
SELECT c.custId, e.empId  
FROM Sales.Customers as c  
CROSS JOIN HR.Employees as e
```

### SQL-89 sintaksa:

```
SELECT c.custId, e.empId  
FROM Saleas.Customers as c, HR.Employees as e
```

SQL-89 sintaksa jeste “kraca”, ali znatno necitljivija. Nema performantnih razlika.

## INNER JOIN

Inner join sadrzi x2 faze procesiranja – Cartesian product izmedju x2 input tabele kao u cross join i onda filtrira redove na osnovu predikata kog specificiras.

Naravno, postoje SQL-89 & SQL-92 standard.

### SQL-92

```
SELECT e.empid, e.firstname, e.lastname, o.orderid  
FROM HR.Employees as e  
INNER JOIN Sales.Orders as o  
ON e.empid = o.empid
```

U sustini matchuje se svaki employee row sa svakim order row-om koji imaju iste employee id's.

Sta se zapravo desava :)

Znaci prvo ide jedan cross join => 9 employee rows x 830 order rows = 7470 rows

Potom join filtrira rows na osnovu “ON” predicate-a e.empid = o.empid sto vraca recimo 830 redova.

ON clause vraca samo redove za koje predicate evaluira na “TRUE”, dakle False i Unknown se odbacuju.

### SQL-89

```
SELECT *  
FROM HR.Employees as e, Sales.Orders as o  
WHERE e.empid = o.empid
```

No performance differences, interpreted as same shit. Jer u sustini desava se cross join pa onda filter uz “ON”. Sad mi je sve jasno.

## BEGIN Joins with extra steps(more join examples)

### Composite join

Composite join je join koji radi filter fazu inner joina na “vise uslova”

Dakle nece biti samo npr ON e.empid = o.empid nego ce biti

ON e.empid = o.empid AND o.propNeki = e.propNeki

Dakle, taj “ON” je kao “WHERE” filter, mozes drugacije stvari da redjas tu.

```
SELECT OD.orderid, OD.productid, OD.qty, ODA.dt, ODA.loginname, ODA.oldval,  
ODA.newval  
FROM Sales.OrderDetails AS OD  
INNER JOIN Sales.OrderDetailsAudit AS ODA  
ON OD.orderid = ODA.orderid AND OD.productid = ODA.productid  
WHERE ODA.columnname = N'qty';
```

## Non-Equi Joins

Kada JOIN condition ukljucuje equality operator, kaze se da je to “equi join”. Kada join condition ukljucuje bilo koji operator sem equality(jednakosti), onda je to non-equi join.

Standard SQL supportuje “natural join” feature koji radi join izmedju x2 tabele na osnovu kolona sa identicnim nazivima. Na primer T1 NATURAL JOIN T2 joinuje redove na osnovu kolona koje imaju iste nazine sa obe strane. T-SQL nema implementaciju natural joina.

Npr za kreiranje unique pairs of employees:

```
SELECT e1.empid, e2.empid  
FROM HR.Employees as e1 INNER JOIN HR.Employees as e2  
ON e1.empid < e2.empid
```

Na primer ovo ce jointovati sve employees koji nemaju isti employee id. Da je samo cross join bio koristen, onda bi rezultat ukljucio i self pairs(1 sa 1) i mirror pairs(1 sa 2 i 2 sa 1).

## Multi join queries

JOIN operator radi na x2 tabele, a kada se pojavi vise tabela onda ce se JOIN izvrsavati redosledom kojim su joinovi napisani.

```
SELECT *  
FROM Customers c INNER JOIN Orders o ON c.custid = o.custid  
INNER JOIN OrderDetails od ON o.orderid = od.orderid
```

Ovde gore ce se desiti sledece:

1. inner join izmedju customers i orders
2. inner join izmedju orderdetails i rezultata prethodnog joina

## END Joins with extra steps(more join examples)

## OUTER JOINS

Outer joins imaju samo x1 standard syntax – onaj sa JOIN & ON clauseom. Postoji samo SQL-92 verzija.

Outer join ima x3 faze. Cartesian product, ON filter & treća faza zvana Adding outer rows koja je unique samo za ovaj tip joina.

U outer join ti obeležavas tabelu kao “preserved table” koriscenjem sledećih keywords:

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

“OUTER” je optional. LEFT keyword znaci da će rows leve tabele biti zadržani. RIGHT keyword znaci da će redovi desne tabele biti zadržani.

FULL OUTER JOIN znaci da su redovi u obe tabele preserved.

Ova treća logicka faza outer joina koja se naziva adding outer rows identificuje redove iz preserved tabele koji nisu pronasli matcheve u ON predicate-u. Potom ih dodaje u result table i koristi “NULL” kao placeholder za attribute iz nonpreserved strane u tim preserved redovima.

```
SELECT c.custid, c.companyname, o.orderid  
FROM Sales.Customer c  
LEFT OUTER JOIN Sales.Orders o  
ON c.custid = o.custid
```

In the query above, all customers will be matched to their respective orders. Those which haven't made any orders will also be shown, but their orderid will be NULL.

U slučaju da nisi znao(znao si, ali ako zaglupis odjednom) – Conceptually prvo se izvršava FROM clause i joinovi u njemu. Ukoliko na tvoj query dodas i WHERE clause, taj WHERE će se izvršiti nakon sto su svi table operatori izvršeni i on ce filtrirati samo end result.

### Including missing values

Na primer imas scenario u kom zelis da filtriras sve porudzbine od 1. jan 2020 do 31. dec 2022. U tim datumima ne zelis samo da vratis dane kada si imao porudzbine nego i dane kada nisi imao porudzbine, znaci kada i orderdate-a nema.

Da bi se solveovalo problem, mozes napisati query koji vraca sekvencu svih datuma u requested periodu i onda odradis outer left join izmedju tog skupa i orders tabele.

Da bi dobio sekvencu datuma, ono sto je lik uradio jeste napravio tabelu sa brojevima – sekvence npr.

```
SELECT DATEADD(day, n-1, CAST('20200101' as DATE)) as orderdate  
FROM dbo.Nums  
WHERE n <= DATEDIFF(day, '20200101', '20221231')+1  
ORDERBY orderdate
```

Potom ces doci i extendovati taj query sa orderima

```
SELECT DATEADD(day, n-1, CAST('20200101' as DATE)) as orderdate,  
      o.orderid, o.custid, o.empid  
  FROM dbo.Nums  
  LEFT OUTER JOIN Sales.Orders o  
    ON o.orderdate = DATEADD(day, n-1, CAST('20200101' as DATE))  
 WHERE n <= DATEDIFF(day, '20200101', '20221231')+1  
 ORDERBY orderdate
```

dbo.Nums je u sustini tabela sa brojevima 1-1000 npr. Od SQL Server 22 mozes koristiti GENERATE\_SERIES funkciju umesto ovog workarounuda.

### Using outer joins in a multi-join query

Dakle, znamo da table operatori se izvrsavaju redosledom kojim su napisani. Zato pazi koju stranu perserveujes. Takodje imaj na umu da outer rows za atribute iz nonperserved strane imaju “null” i samim tim UNKNOWN se odbacuje u ON filteru.

### Filtering attributes from the nonperserved side of an outer join

Uzeti u obzir sledeci query:

```
SELECT *  
  FROM Sales.Customers c  
  LEFT OUTER JOIN Sales.Orders o ON o.custid = c.custid  
 WHERE o.orderdate >= '20220101';
```

Dakle ovde se prvo vraca svi customers sa njihovim porudzbinama, ali onda posto cemo imati left join gde se vrate i customeri bez porudzbina, a mi ukjucujemo WHERE filter na osnovu vrednosti koja ce biti NULL za customere koji nemaju porudzbine, mi cemo u sustini imati kao neki rezultat iz INNER JOIN-a. Oni sa NULL orderdate-om ce rezultovati u UNKNOWN. Ovo je cest problem, bolje razmisljaj :)

## Using COUNT aggregate with outer joins

Drugi cesti bug jeste koriscenjem COUNT sa outer joinovima. Kada gripises rezultat outer joina i koristis COUNT(\*) aggregate, taj aggregate ukljucuje obe inner rows i outer rows jer prebrojava redove bez obzira na njihov sadrzaj.

Uglavnom ne bi trabalo da ukljucujes outer rows into consideration u svrhe brojenja.

Na primer treba da prebrojavas broj porudzbina svakog customera

```
SELECT c.custid, COUNT(*) as numoforders  
FROM sales.customers c  
LEFT OUTER JOIN sales.orders o ON c.custid = o.custid  
GROUP BY c.custid
```

Customeri 22 i 57 koji nemaju ordere ce imati outer row u rezultati joina pa samim tim ce im numoforders reci "1".

Zato umesto COUNT(\*) ides COUNT(o.orderid) jer COUNT(NULL) je ignorisan, a outer row to nece imati.

## Exercises sve do 149. stranice

### Subqueries

SQL Supportuje pisanje query-a unutar query-a tj nesting queries.

"Najspoljnii"(outer most) query je taj koji ce vratiti krajnji rezultat. Inner query se evaluira u run time i npr rezultat moze da se menja u zavisnosti od podataka, jelte. Koristi se umesto da sad hardkodiras neku konstantu.

Sub-query moze biti:

- self contained
- correlated

Self contained subquery nema zavisnosti od tabela spoljnog query-a. Correlated ima dependency tabela spoljnog querya.

Subquery result moze biti single valued, multivalued ili table valued.

Ovaj chapter se vise fokusira na single valued subqueries(scalar queries) i multivalued subqueries.

## Self contained subqueries

Self contained subqueries su nezavisni query-i od outer query-a.

Logicki redosled izvrsavanja je takav da se prvo izvrsava subquery kod, a tek onda outer query. Potom outer query koristi rezultat subquery-a.

Na primer zelis da uzmes najskuplji proizvod i zapamti ga.

```
DECLARE @priciestItem INT = (SELECT MAX(price) FROM Products)
```

ili umesto da ga pamti

```
SELECT * FROM Products WHERE price = (SELECT MAX(price) FROM Products)
```

Scalar subquery mora da returnuje single value. Ako returnuje vise vrednosti onda ce failovati u toku runtime.

Zasto ce failovati ako ne vrati skalarni rezultat? Pa jer equality je binarni operator koji poredi jednu vrednost sa leve sa jednom vrednostii s desne strane. Da se vrati vise vrednosti, puca.

Ukoliko subquery ne vrati nikakvu vrednost, onda se taj prazan rezultat konvertuje u **NULL!**

Poredjenej sa NULL yielduje u UNKNOWN i taj query filter potom nece vratiti rezultat jer samo oni koji evaluiraju na TRUE vracaju.

## Self contained multivalued subquery

Isti k samo ces raditi sa npr **IN** predicateom

```
SELECT * FROM Products
```

```
WHERE productId IN (SELECT productId FROM Products WHERE price > 200)
```

Subquery nekad moze da ima bolji performans od joina. Nekada join moze imati bolji performans. Underlying DB engine ce svakako dodati "optimizacione tehnike" pa ne mozes uvek 100% biti siguran.

Pristup koji ima covek koji je napisao knjigu je sledeci: Prvo napisi intuitivni query koji daje tacan rezultat. Potom podesavaj performans :)

Poslednji primer:

Treba da napises query koji vraca sve incomplete orderid's koji fale izmedju minimum i maximum ones u tabelama.

```
SELECT n FROM dbo.Nums
WHERE n BETWEEN
    (SELECT MAX(orderid) FROM Orders) AND
    (SELECT MAX(orderid) FROM Orders)
AND NOT IN (SELECT orderid FROM CompletedOrders)
```

## Correlated subqueries

Correlated subqueries su subqueris koji referuju atributima spoljnih query-a u kojima su nestovani. To znaci da je subquery zavistan od outer query i ne moze biti invoked kao standalone query.

Logicki gledano, subquery se evaluira separately za svaki outer row u logical query processing step u kom se pojavljuje.

Primer: kod koji vraca najskuplje order za customera(probaj i sa group by)

```
SELECT o1.custid, o1.orderid
FROM Orders o1
WHERE orderid = (SELECT MAX(o2.orderid) FROM Orders o2 WHERE o2.custid = o1.custid)
```

TLDR: Subquery gets executed for each row in the outer query.

E sada, mnogo je teze troubleshootovati ovakve queryje. Nazalost. Ne mozes samo doci i executeovati subquery jer vrednost zavisi od outer query. Zato sta se radi je sledece:

Za testing subquery, uzmes jednu vrednost(red) iz outer query-a i hardkoridas u inner query.

Na primer: SELECT MAX(o2.orderid) FROM Orders o2 WHERE o2.custid = 86

Za red sa customerid gde je 86 mozes da vidis sta ce subuery vratiti

Potom uzmes taj rezultat i testiras outer query

```
SELECT o1.custid, o1.orderid
FROM Orders o1
WHERE orderid = 102534
```

Mozes tako da se igras, malo je zeznutno, ali funkcione.

## EXISTS Predicate

EXISTS predicate prihvata subquery kao input i vraca TRUE ako subquery vrati bilo kakav row. Rezultat bez rows vraca FALSE

```
SELECT custid, companyName  
FROM Customers c  
WHERE country = 'Spain'  
AND EXISTS(SELECT * FROM Orders o WHERE o.custid = c.custid)
```

Query iznad vraca customere iz spanije koji su napravili neku kupovinu.

Mozes koristiti i NOT EXISTS koji ce vratiti "true" ukoliko nema redova koji postuju uslov, jelte.

Na primer za ovaj query iznad da si koristio NOT EXISTS onda bi dobio spance koji nemaju porudzbine.

PERFORMANCE EXISTS QUERY-a; Dakle DB engine shvata da je dovoljno vratiti jedan rezultat za EXISTS query. Shvati to kao short-circuit evaluation. Same applies na IN Predicate. Cak i za (\*) koje je smatrano losom praksom, EXISTS je okej. Predicateu je stalo samo do postojanja matching redova bez obzira sta napises u SELECT tako da ce db engine ignorisati subquery-s select list.

To ne znaci da (\*) nece biti resolveovan u EXISTS(SELECT \* FROM...), ali performance differene sa necim poput EXISTS(SELECT 1 FROM...) je bukvalno neprimetno.

Takodje EXISTS koristi two valued logic, a ne three valued logic.

## Returning previous or next values

Recimo da moras queryati Orders table i vratiti za svaki order informacije o trenutnom orderu i prethodni orderid. Taj deo "previous" implicira redosled kog tabele nemaju, jelte.

Jedan pristup ovome jeste "maksimalna vrednost koja je manja od trenutne vrednosti."

```
SELECT orderid, orderdate, empid, custid,  
(SELECT MAX(o2.orderid)  
FROM Orders o2 WHERE o2.orderid < o1.orderid) as prevorderid  
FROM Orders o1
```

## Using running aggregates

Running aggregates su agregati koji akumuliraju vrednosti na osnovu nekog redosleda.

orderyear	qty	runqty
2020	9581	9581
2021	25489	35070
2022	16247	51317

Recimo da moras da izracunas zbir svih “quantity” vrednosti do neke goine.

Za 2021 to ce biti quantity iz 2020 i 2021. Za 2020 samo iz 2020.

```
SELECT orderyear, qty,
(SELECT SUM(o2.qty) FROM OrderTotalsByYear o2
WHERE o2.orderyear < o1.orderyear) as runqty
FROM OrderTotalsByYear o1
ORDER BY orderyear
```

- ⇒ Ne kontam sta je fora sa ovim. Ovo je obican correlated query gde se taj correlated query izvrsava u SELECT(skoro poslednjoj fazi) clauseu

## BEGIN - Dealing with misbehaving subqueries

### Null Trouble

Uzmi query koji ce vratiti customere koji nemaju ordere:

```
SELECT * FROM Customers c WHERE custid NOT IN (SELECT o.custid FROM Orders o)
```

I dobijes neki rezultat, set od x2 row-a.

custid	companyname
22	Customer DTDMN
57	Customer WVAXS

E sada, odradi insert u orders tabelu gde ce jedan custid imati vrednost “NULL”

Rezultat ce biti prazan skup, tj nista neces dobiti. Kako to.

Jer jebeni three valued logic.

Sta se desava?

Posto subquery vraca multiresult u taj “IN” nad kojim imas negaciju sa “NOT”, desi se da ukoliko se pojavi NULL u customer id orders, ne mozes sa preciznoscu reci da li jeste ili nije taj customer id tu. Jer NULL predstavlja odsustvo vrednosti, jelte.

Da imas non null multivalue u IN onda mozes sa sigunoscu reci da li je neki custid u tom spisku ili nije.

IN predicate returnuje UNKNOWN za customera koji nije napravio order(custid: 22) jer ce se porediti sa NULL u jednom momentu i dobices UNKNOWN. FALSE ili UNKNOWN yielduje u UNKNOWN.

Kada se to prevede u "vrednost" ti taj NOT IN(1, 2, ..., NULL) u momentu dobijes NOT UNKNOWN sto je => UNKNOWN

TLDR => NULL u "IN" operatoru ce uvek rezultirati u UNKNOWN

Resenje:

NOT EXISTS (SELECT \* FROM Orders o WHERE o.custid = c.custid)

ili

NOT IN (SELECT o.custid FROM Orders o WHERE o.custid IS NOT NULL)

### Substitution errors in subquery column names

Lik je dosao i rekao "e pazi na typo"

### END – Dealing with misbehaving subqueries

Exercises sve do 177. stranice

### Derived Tables

Derived tabele su definisane u FROM clause u outer query-u. Njihov scope postojanja jeste outer query tako da cim se outer query izvrsi, one prestaju da postoje.

```
SELECT *
FROM (
    Select custid, companyname
    from sales.customers where country = 'usa') as USACustomers
=> specificiras query koji predstavlja tabelu, posle toga dodas "AS" gde kazes derived table name.
```

**Da bi expression bio ispravan, query mora ispuniti sledece zahteve:**

- Da nije u SNS
- Redosled row-ova da nije zagarantovan => ORDER BY je dozvoljen samo ako ne sluzi za prezentacionu svrhu. Znaci ako koristis OFFSET-FETCH npr.
- Sve kolone moraju imati nazive
- Svi nazivi kolona moraju biti unique

## Assigning column aliases

Recimo scenario u kom zelis da grupises po alias column name

```
SELECT YEAR(orderdate) as orderyear  
FROM Orders  
GROUP BY orderyear
```

Ovde ces dobiti error da orderyear ne postoji, jelte.

Resenje: derived table

```
SELECT * FROM (  
    SELECT YEAR(orderdate) as orderyear  
    FROM Orders) as xyz  
ORDER BY orderyear
```

Nema performance benefita ni gubitaka. MSSQL ce u pozadini izvrsiti nesto poput sledeceg:

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcunds  
FROM Sales.Orders  
GROUP BY YEAR(orderdate);
```

Table expressions se uglavnom koriste iz logickih razloga. Ne iz performantnih razloga.

## Using arguments

Argumenti, upoznat si sa njima. Dakle mogu biti lokalni argumenti ili argumenti neke routine poput stored procedure ili funkcije.

```
DECLARE @var_name AS var_type = var_value  
  
DECLARE @custid AS INT = 5;  
SELECT * FROM Orders WHERE custid = @custid;
```

AS je optional.

## Nesting

Mozes nestovati

```
SELECT * FROM (  
    SELECT kita, kita2 FROM (  
        SELECT ckita, ckita2 FROM orders WHERE ckita = 22, ckita2 < 55) as nest 2) as nest1
```

Problem, citljivost nije ni za kurac

## Multiple references

Jos jedan problematican aspekt derived tabela jeste u slucajevima kada trebas da joinujes vise instanci istih. JOIN tretira dva inputa kao skupovi gde jelte, skup nema order.

To znaci da ako definisesh derived table i aliasujes ga kao input joina, ne mozes referovati isti alias u drugi input joina.

Moras

```
SELECT *
FROM (inner_query) as cur
LEFT JOIN (that_same_inner_query) as prv
```

## Common table expressions

Common table expressiosn CTEs su jos jedan standard form table expressiona slican deriviranim tabelama, ali sa nekoliko bitnih “advantages”(zaboravih rec).

Definisane su koriscenjem **WITH statementa**

```
WITH <CTE_NAME>[(target_column_list)]
AS
(<inner_query_defining_CTE>
<outer_query_against_CTE>
```

Inner query mora pratiti sve isto kao i derived table expression.

```
WITH USACusts AS
(
    SELECT custid, companyname
    FROM Sales.Customers
    WHERE country = 'USA'
)
SELECT * FROM USACusts;
```

Kao i sa derived tabelama, cim se outer query zavrzi tako ce i ta tabela biti obrisana.

## Assigning column aliases in CTEs

CTEs podrzavaju dva vida column aliasinga. Inline i external.

Za inline formu, to se specificira unutar samog selecta u common expressionu.

Sto se tice external formu, to se specificira kao lista kolona u zagradi odmah nakon CTE imena.

### **Inline:**

```
WITH C AS
(
    SELECT YEAR(orderdate) as orderyear, custid FROM Sales.Orders
)
```

### **External:**

```
WITH C(orderyear, custid) AS
(
    SELECT YEAR(orderdate, custid FROM Sales.Orders
)
```

## Koriscenje argumenata u CTE

Kao i sa derived tabelama mozes koristiti argumente u inner query da bi definisao CTE.

```
DECLARE @empid as int = 3;
```

```
WITH C AS
(
    SELECT * ....WHERE empid = @empid
)...
```

## Defining multiple CTE

```
WITH C1 as
(
    ...
),
WITH C2 as
(
    SELECT ... FROM C2 c2 GROUP BY...
)
SELECT * FROM C1 join C2 on ....
```

Zasto je ovo kul? Pa jer mozes da referenciras vrednosti iz jednog CTE sa drugim CTE. Ne moras ih nestovati, samo ih razdvojis kroz razmak. Svaki CTE moze referencirati prethodni CTE i mozes queryovati sve definisane CTE's.

## Multiple references in CTEs

Cinjenica da se prvo definise CTE ti dozvoljava da u outer query-u saljes upit ka vise instanci istog CTE-a

```
WITH C AS  
(...)  
SELECT * FROM C as c1 JOIN C as c2 ON ...
```

## Recursive CTEs

CTEs imaju podrsku za rekurzije. Rekurzivni CTE su definisani by the SQL standard i definisan je na osnovu barem x2 query-a(more are possible) – gde je bar jedan query anchor member i bar jedan recursive member.

Basic rekurzija izgleda nekako ovako:

```
WITH <CTE_Name>[(<columns>)]  
AS  
(  
<anchor_member>  
UNION ALL  
<recursive_member>  
)  
<outer_query_against_CTE>
```

**Anchor member** je query koji vraca validni relacioni result table – kao query koji sluzi za definisanje nerekurzivnog table expressiona. On se invokeuje samo jednom.

**Recursive member** jer query koji ima reference ka CTE name i invoked je sve dok se ne vrati prazan skup. Referenca ka CTE name predstavlja prethodni result set.

Prvi put pri izvrsavanju rekurzivnog membera, prethodni result set predstavlja ono sto je anchor member vratio. Pri svakom sledecem izvrsavanju rekurzivnog clana, referenca na CTE predstavlja result set kog vrati prethodno izvrsenje rekurzivnog membera.

CTE name represents:

- 1<sup>st</sup> invocation => anchor member result
- 2<sup>nd</sup> invocation => recursive member result
- 3<sup>rd</sup> invocation => recursive member result
- ... => until empty set

Oba queryja moraju biti kompatibilna u smislu broja kolona i tipa podataka koje vracaju. Referenca na CTE name u outer query predstavlja unifikovani rezultat skupova invokacija anchor membera i SVIH invokacija recursive membera.

Recimo sada u praksi -> Zelis da vratis informacije o zaposlenom i svim njegovim subordinates na svim nivoima(direct ili indirect).

```
WITH EmpsCTE AS
(
SELECT empid, mgrid, firstname, lastname
FROM HR.Employees
WHERE empid = 2

UNION ALL

SELECT C.empid, C.mgrid, C.firstname, C.lastname
FROM EmpsCTE as P
INNER JOIN HR.Employees AS C ON C.mgrid = P.empid
)
SELECT rmpid, mgrid, firstname, lastname
FROM EmpsCTE;
```

Anchor member queryuje za employee sa ID od 2.

Recursive member uzima i joinuje od tog membera, sve druge zaposlene kojima je on menadzer. Potom svi oni kojima je on menadzer se uzimaju i gleda se kome su oni menadzer.

U slucaju logicke greske u JOIN predicateu u rekurziji ili ako ima problema sa podacima koji rezultuju u "ciklusima", recursive member moze biti invokeovan beskonacno puta(endless loop). Kao safety measure, sql server restricts broj rekurzivnih invokacija na 100 po defaultu.

Code ce failovati ako je recursive member invokeovan vise od 100 puta. Mozes default ponasanje promeniti sa OPTION(MAXRECURSION n) na kraj outer query-a gde je n integer od 0-32767.

Ako zelis da uklonis ogranicavanje izvrsenja rekurzivnog querya, onda stavi MAXRECURSION 0. SQL Server stores medjurezultate izvrsavanja izmedju anchor i rekurzive membera unutar tempdb. Ako removeujes restriction i imas runaway query, work table ce se poprilibno napuniti i query se nikada nece zavrsiti.

## Views

Derived statements & CTE's imaju single statement scope i nisu reusable. Views & inline table-valued functions(Inline TVFs) su x2 tipa table expressiona cije se definicije cuvaju kao objekti u bazi pa su reusable.

U vecini slucajeva, views i TVFs se tretiraju kao derived tables i CTEs.

```
CREATE OR ALTER VIEW Sales.USACusts
```

```
AS
```

```
SELECT custid, companyname, ...
```

```
FROM Sales.Customers
```

```
WHERE country = 'USA';
```

```
GO
```

**GO command** se ovde koristi da bi terminateovala "batch" u T-SQL. Bice kasnije objasnjeno.

CREATE OR ALTER syntax kreira ili alteruje definiciju objekta. CREATE VIEW ce samo kreirati npr.

```
SELECT * FROM Sales.USACusts;
```

## Views & Order by

Query koji koristis da bi definisao view mora ispostovati sve sto mora i inner query. Znaci bez orderinga, sve view kolone moraju imati ime i imena moraju biti unique.

Uz TOP ti mozes u sustini koristiti ORDER BY i ljudi misle da ces kreirati "ordered view". Iako je code tehnicki ispravan i view je kreiran, treba da znas da ako outer query nad tvojim viewom nema ORDER BY clause, prezentacioni order nije zagarantovan.

Cak iako ti izgleda kao da je ordered, to mozda moze samo da ti se cini jer je takva pozicija podataka.

## View options

Kada kreiras ili alterujes view, mozes specificirati view attributes & options kao deo view definitiona.

Unutar headera viewa pod WITH clause, mozes specificirati atribute poput:

- ENCRYPTION
- SCHEMABINDING

Na kraju querya mozes specificirati:

- WITH CHECK OPTION

## **ENCRYPTION Option**

Ova opcija je dostpuna kada kreiras ili alterujes view, stored procedures, triggers & user defined functions.

Encryption indicates da SQL Server ce interno storeovati text u definiciji objekta u “obfuscated formatu”. Taj obfuscated text nije direktno vidljiv koristnicima kroz bilo koji catalog objekata, samo privileged useri mogu da gledaju.

Recimo imas:

**CREATE OR ALTER VIEW Sales.USACusts AS**

**SELECT custid, companyname, contactname, contacttitle, address, city, region, postalcode, country, phone, fax FROM Sales.Customers WHERE country = N'USA';**

**GO**

Ukoliko pokusas da pribavis definiciju objekta na sledeci nacin:

**SELECT OBJECT\_DEFINITION(OBJECT\_ID('Sales.USACusts'))**

Dobices rezultat. Da si postavio ENCRYPTION opciju, ne bi dobio output.

Da bi postavio opciju koja se postavlja u headeru onda

**CREATE OR ALTER VIEW <view\_name> WITH ENCRYPTION**

**AS**

**SELECT \***

**...**

**GO**

I ukoliko sada ponovo pokusas da dobijes definiciju objekta, onda dobijes “NULL”

Bilo koja alternativa object definition funkciji ce ti dati neki error ili nista.

## **SCHEMABINDING Option**

Dostupna je Viewovima, UDFs i natively compiled modulima; binduje schema referenced objekta i kolona u schema referencing objekta.

Naznacava da referenced objects ne mogu biti dropped i da referenced columns ne mogu biti dropped ili promenjene.

**CREATE OR ALTER VIEW <> WITH SCHEMABINDING**

**AS**

**SELECT \*, address,... FROM Sales.Customers WHERE <>**

**GO**

I sada ako pokusas da dropujes **address** kolonu iz **Sales.Customers** dobijas error:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

You get the following error:

```
Msg 5074, Level 16, State 1, Line 345  
The object 'USACusts' is dependent on column 'address'.  
Msg 4922, Level 16, State 9, Line 345  
ALTER TABLE DROP COLUMN address failed because one or more objects access this column.
```

Bez schemabindinga bi mogao lagano dropovati. Sa schemabindingom ces izbeci runtime greske u kojima ces queryovati kolone tabele koje ne postoje u slucaju izmene podataka.

Da bi se supportovalo schemabinding, postoje pravila:

- Bez koriscenja \* u SELECT -> izlistaj sve column names
- moras koristiti schema qualified two part names

## CHECK Option

Sluzi kako bi sprecio modifikacije kroz view koji bi mogle uticati na inner query filter tog view-a. Npr query koji definise view USACusts filtrira customere na osnovu drzave da je United States.

Kada to uradis bez check-a, to znaci da mozes insert kroz view customera iz drugih zemalja i mozes update country postojecih customera kroz taj view.

Npr sledeci kod ce dodati customera iz US-a kroz view.

```
INSERT INTO Sales.USACusts(companyname, contactname, ...)  
VALUES ('customer abcde',...), ('customer_2 abcde',...)
```

Row je insertovan kroz view unutar customers table.

E sada, posto si uneo x2 customera kroz view koji npr nisu imali US kao country nego recimo **UK**, onda kada budes izvrsio view neces ih dobiti nazad.

```
SELECT custid, companyname, country  
FROM Sales.USACusts  
WHERE companyname LIKE 'customer%';
```

Kako bi odradio prevenciju modifikacija koji se conflictuju sa view's filter, dodajes WITH CHECK OPTION na kraj quarya koji definise view.

```
CREATE OR ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS
SELECT *
FROM Sales.Customers
WHERE country = 'USA'
WITH CHECK OPTION;
GO
```

I sada kada pokusas da insertujes nesto sto se conflictuje sa viewovim filterom, dobijas error.

### Inline Table-Valued Functions

Inline TVFs su reusable table expressions koje podrzavaju input parametre. TVF's su slicne viewovima pa se nekad i referuju kao "views with inputs", ali ne formalno.

T-SQL podrzava i multi-statement TVF koji populateuje i returnuje table variable. To nije table expression jer se ne bazira na queryu

```
CREATE OR ALTER FUNCTION dbo.GetCustOrders
    (@cid AS INT) RETURNS TABLE
AS
    RETURN
    SELECT orderid, shippeddate, custid, empid, orderdate, ...
    FROM Sales.Orders
    WHERE custid = @cid;
GO
```

Inline TVF's header ima mandatory RETURNS TABLE clause sto znaci da funkcija vraca table result. Takodje i mandatory RETURN clause pre inner query-a.

Tu je i input parametar.

```
SELECT orderid, custid FROM dbo.GetCustOrders(1) as O
```

MAKE SURE YOU PROVIDE AN ALIAS!

Nije uvek requirement, ali jako dobra praksa i imas manje sansi za errore.

Takodje mozes TVFs mogu biti naravno i deo JOIN-a.

```
DROP FUNCTION IF EXISTS dbo.GetCustOrders;
```

## The APPLY Operator

Apply operator je table operator koji se koristi u FROM clause-u querya. Postoje x2 podrzana tipa apply-a: CROSS APPLY & OUTER APPLY

APPLY performs its work in logical query phases kao i JOIN. CROSS APPLY implements samo x1 logical query processing phase dok OUTER APPLY implements x2 phases.

APPLY nije standard. Njegov SQL Standard counterpart se naziva **LATERAL**, ali standardna forma nije implementirana u SQL Serveru.

APPLY operator radi nad x2 tabele, left & right(nek ih tako nazovemo). Right table je cesto derived table ili TVF. CROSS APPLY primenjuje right table na svaki row leve tabele i kreira result table sa unifikovanim rezultatima.

Slican je cross joinu, istina i naredna x2 querya ce vratiti isti rezultat:

```
SELECT S.shipperid, E.empid  
FROM Sales.Shippers AS S CROSS JOIN HR.Employees AS E;
```

```
SELECT S.shipperid, E.empid  
FROM Sales.Shippers AS S CROSS APPLY HR.Employees AS E;
```

Join tretira x2 inputa kao skupove pa stoga ne postoji order izmedju njih. Sa APPLY leva strana se prva evaluira i potom se desna strana evaluira za svaki red leve tabele.

Right side moze imati reference elementima sa leve. Te reference su u ustini correlations.

Na primer da vratis x3 most orders za svaki customer:

```
SELECT *  
FROM Sales.Customers AS C  
CROSS APPLY (SELECT TOP (3) orderid FROM Sales.Orders AS O WHERE O.custid = C.id  
ORDER BY orderdate DESC, orderid DESC) AS A
```

Ukoliko leva strana nema rezultata, onda krajnji rezultat ce biti "nista". Znaci nema na cemu da se applyuje desna tabela. Ukoliko leva strana ima rezultata, ali nema na cemu da se matchuje tj derived table je empty set, onda takodje se nista ne returnuje.

Ukoliko na primer zelis da bez obzira na rezultat desne strane vratisrowove sa leve strane cak iako nema matchova, onda koristi OUTER APPLY.

Druga logical faza ovo operatora zadrzava sve leve row-ove.

Ona zadrzava row-ove sa leve strane za koje nije bilo matcheva sa desne strane i koristi

NULLs kao placeholdere. U sustini je skoro ekvivalentno sa LEFT OUTER JOIN. Ne postoji ekvivalent za RIGHT OUTER JOIN zbog prirode APPLY-a.

Tako da npr kombinacija za TVFs i CROSS APPLY bi za top 3 rows bi izgledalo ovako:

```
CREATE OR ALTER FUNCTION dbo.TopOrders
    (@custid AS INT, @n AS INT)
RETURNS TABLE
AS
RETURN
    SELECT TOP (@n) orderid, empid, orderdate, requireddate
    FROM Sales.Orders
    WHERE custid = @custid
    ORDER BY orderdate DESC, orderid DESC;
GO
```

You can now substitute the use of the derived table from the previous examples with the new function:

```
SELECT
    C.custid, C.companyname,
    A.orderid, A.empid, A.orderdate, A.requireddate
FROM Sales.Customers AS C
CROSS APPLY dbo.TopOrders(C.custid, 3) AS A;
```

## Exercises do 210. stranice

### SET Operators

Set operatori(operatori skupova) su operatori koji kombinuju redove iz dva querya koji davaju skupove kao rezultate. Neki operatora otklanjaju duplike iz tih rezultata pa vracaju skup, dok drugi bas i ne pa vracaju multiset.

To su: UNION, UNION ALL, INTERSECT, EXCEPT.

#### **Input query1**

**<set\_operator>**

#### **Input Query2**

**[ORDER BY ...];**

Kako set operator radi sa skupovima, tako rezultati query-a ne smiju biti ordered. Query sa order by vraca ordered result, a ne multiset.

Ovar orderby se primenjuje na sam rezultat set operatora, to smes.

Dva input querya moraju vratiti rezultat sa istim brojem kolona i svaka kolona mora da ima podudarajuce tipove podataka. To znači "taj tip podataka" ili da može da se implicitno konvertuje u higher data type. Uz cast/convert možes konvertovati naravno.

Imena kolona krajnjeg rezultata zavise od 1. query-a tako da pisces aliase u 1. query ako ti treba. Best practice je da sve kolone u svim query-ima imaju isti naziv.

Kada set operator poredi dva reda izmedju x2 inputa on koristi distinctness-based comparison, a ne equality base comparison. Taj distinctness-based comparison je isti kao u distinct predicate-u koji tretira NULL kao non-null value za comparison purposes gde jedan NULL se ne razlikuje od drugog NULL-a, a NULL se razlikuje od vrednosti broja 13 npr.

## UNION Operator

Unifikuje x2 input query-a. Ako se row pojavljuje u jednom skupu, pojavice se i u rezultatu.

T-SQL supports UNION ALL & UNION(implicit DISTINCT) flavors UNION operatora.

```
SELECT country FROM HR.Employees  
UNION  
SELECT country FROM HR.Employees
```

Ce vratiti **m** broj rezultata. DISTINCT se implicitno primenjuje i duplikati se otklanjaju.

## UNION ALL Operator

Unifies two input query results bez da pokusa da otkloni duplike iz rezultata.

Znaci da

```
SELECT country FROM HR.Employees  
UNION ALL  
SELECT country FROM HR.Employees
```

Ce vratiti **m+n** broj rezultata

## INTERSECT Operator

Ovo je u sustini presek skupova. Ovaj operator vraca samo redove koji se pojavljuju u **query\_1** i **query\_2**

Sam INTERSECT naravno ima implicit DISTINCT.

## INTERSECT ALL Operator

U sustini postoji u SQL standardu, ali nije implementiran u T-SQL.

Alternativa:

Znaci “ALL” predstavlja da vraca sve redove bez da se otklanjaju duplikati. U slucaju INTERSECT ALL to znaci da ce se vratiti svi redovi iz preseka, ukljucujuci i duplike.

Koriscenjem ROW\_NUMBER funkcije mozes brojati occurrences svakog reda u svakom input query-u. U sustini specificiras sve atribute u PARTITION BY clause funkcije i koristis (SELECT <any constant>) u ORDER BY clauseu funkcije.

SELECT

```
ROW_NUMBER() OVER(PARTITION BY country, region ORDER BY (SELECT 0)) AS rownum,  
country, region FROM HR.Employees
```

INTERSECT

```
ROW_NUMBER() OVER(PARTITION BY country, region ORDER BY (SELECT 0)) AS rownum,  
country, region FROM Sales.Customers
```

rownum	country	region	city
1	UK	NULL	London
1	USA	WA	Kirkland
1	USA	WA	Seattle
2	UK	NULL	London
3	UK	NULL	London
4	UK	NULL	London

Da bi otklonio **rownum** to mozes sa WITH statementom

**WITH INTERSECT\_ALL**

**AS (intersect\_query\_od\_gore)**

**SELECT <sve\_sem\_rownum> FROM INTERSECT\_ALL**

## EXCEPT Operator

Operator koji implementira “minus” iliti razliku skupova.

U sustini vraca redove koji se pojavljuju u PRVOM QUERY-U, ALI NE I U DRUGOM. Ovo je prvi set operator u kom redosled deklarisanja operatara postaje bitan jer jelte, samo redovi iz 1. query-a se vracaju.

## EXCEPT ALL Operator

Pazi sada ovo.

U teoriji kada imas neki red R koji se pojavljuje u query1 i red X koji je njegov corresponding row u query2. Ukoliko je broj redova R vezi od broja redova X, EXCEPT ALL ce vratiti sve one redove R koji nemaju svoj corresponding occurrence u drugom query-u.

Recimo imas x10 R redova i x9 X redova. To znaci da ces dobiti x1 R red u rezultatu :)

T-SQL naravno ne nudi built in EXCEPT ALL operator, ali alternativa je takodje sa ROW\_NUMBER

```
WITH EXCEPT_ALL
AS
(
    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
            ORDER      BY (SELECT 0)) AS rounum,
        country, region, city
    FROM HR.Employees

    EXCEPT
```

[REDACTED]

```
    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
            ORDER      BY (SELECT 0)),
        country, region, city
    FROM Sales.Customers
)
```

```
SELECT country, region, city
FROM EXCEPT_ALL;
```

## Precedence

SQL definise "precedence" izmedju skupovnih operacija.

Ides:

INTERSECT > UNION > EXCEPT

Koriscenjem ALL varijante nista ne menja. Ukoliko se u upitu koristi vise skupovnih operacija, prvo se evaluiraju INTERSECT pa onda svi union, pa onda svi except.

```
SELECT * from q1
```

```
EXCEPT
```

```
SELECT * from q2
```

```
INTERSECT
```

```
SELECT * from q3
```

Prvo se evaluira q2 i q3 iako je INTERSECT posle njih. Potom ide EXCEPT. Najveci precedence u svemu imaju zagrade pa njih mozes koristiti kako bi kontrolisao redosled operacija.

## Circumventing unsupported logical phases

Query-i koji ucestvuju u skupovnim operacijama smeju da sadrže sve klazule sem ORDER BY.

Jedina klazula koja je dozvoljena nad rezultatom skupovne operacije je ORDER BY. :)

Mozes koristiti WITH ili inline query kako bi primenjivao i druge clauses nad rezultatom skupovne operacije.

Exercises do 230. stranice

## T-SQL For Data Analysis

Ovaj deo se fokusira na T-SQL for data analysis

### Window functions

Window function je fn koja za svaki row computeuje scalar result na osnovu kalkulacija nad podskupom redova underlying query-a.

Subset of rows(podskup redova) se naziva *window* i bazira se na window descriptoru koji se odnosi na trenutni red.

Sintaksa window functiona koristi clause **OVER** u kom se provideuju window specifikacije.

Primer neceg slicnog su SUM/COUNT/AVG, slucaj kada imas skup i kada trebas da vratis jedan(skalarni) rezultat za ceo skup.

U slucaju grouped query-a gubis nesto bitno – detalje. Dobijas jelte agregirane informacije, ali nakon sto grupises redove, sve komputacije u upitu moraju da se izvrsavaju nad definisanim grupama.

Cesto moraces izvrsavati kalkulacije koje uključuju detail i aggregate elemente. Window function nisu tu ogranicene.

Window function je evaluated per detailed row i primenjuje se nad subset of rows koji potice iz query result seta.

Rezultat window funkcija je skalarna vrednost koja se dodaje u drugu kolonu u query rezultu. Unlike grouped functions, windows functions don't cause you to lose detail.

Recimo da zelis da vratis vrednost porudzbina i procenat koji svaka porudzbina zauzima od totalne sume porudzbina nekog kupca. Window funkcijom mozes vratiti customer total, zajedno sa detaljima redosleda vrednosti i mozes i computeovati procenat trenutne porudzbine naspram totalne sume svih porudzbina.

Sto se tice subqueries(self contained/correlated) – njih mozes koristiti da primenis scalar agregat calculation nad skupom, ali njihova pocetna tacka je svez pogled nad podacima, rather than the undelying query result set.

Ako underlying query ima table operators, filtere i druge elemente upita, oninece uticati sta subquery vidi. Ako zelis da subquery “gleda” na underlying query, moras ga ponoviti.

Kada su u pitanju WINDOW FUNCTIONS, window function se primenjuje nad podskupom redova underlying query result set-a. Ne nad svezim pogledom na podatke. Tako da, sve sto dodas u underlying query ce se automatski primeniti na sve window funkcije koje se koriste u query-u. Naravno, mozes restrictovati window ako zelis.

Jos jedan benefit window funkcija je sto mozes definisati order kao deo specifikacije kalkulacije sto ne prkosi relacionom pogledu na rezultate. Redosled(order) ovde se definise kao kalkulacija i ne treba se mesati sa prezentacionim smislom redosleda.

```
SELECT empid, ordermonth, val,  
SUM(val) OVER(PARTITION BY empid  
              ORDER BY ordermonth  
              ROWS BETWEEN UNBOUNDED PRECEDING  
                     AND CURRENT ROW) AS runval  
FROM Sales.EmpOrders;
```

Ovo je primer querya nad Sales.EmpOrders koji ce koristiti window aggregate fn da computeuje running total values za svakog employeea i mesec.

empid	ordermonth	val	runval
1	2020-07-01	1614.88	1614.88
1	2020-08-01	5555.90	7170.78
1	2020-09-01	6651.00	13821.78
1	2020-10-01	3933.18	17754.96
1	2020-11-01	9562.65	27317.61
...			
2	2020-07-01	1176.00	1176.00
2	2020-08-01	1814.00	2990.00
2	2020-09-01	2950.80	5940.80

Postoje x3 dela u definiciji window funkcije

OVER -> window-partition clause, window-order clause & window-frame clause

Prazan “OVER()” predstavlja ceo underlying query’s result set. Onda sta god da dodas window specifikaciji ce u sustini restrictovati window.

**Window-partition clause(PARTITION BY)** restrictuje window na subset redova koji imaju istu vrednost u particionim kolonama kao trenutni red. U gore primeru radilo se particionisanje na osnovu *empid*. Za underling row sa empid 1, window exposeuje function filteru samo one redove gde je employee id = 1.

**Window-order clause(ORDER BY)** definise redosled(ali ne prezentacioni redosled). U window ranking funkciji, window ordering daje znacaj ranku. U nasem primeru, ordering se desava na osnovu *ordermonth* kolone.

**Window-frame clause(ROWS BETWEEN <top delimiter> AND <bottom delimiter>)** – filtrira frame ili podskup redova iz window particije izmedju dve specificirane vrednosti. U nasem primeru je frame definisan bez low boundary pointa (UNBOUNDED PRECEDING) i extenduje se do trenutnog reda(CURRENT ROW). In addition to the window-frame unit ROWS, postoji i RANGE, ali nije skroz implementirano u T-SQL.

NOTE: Posto pocetna tacka window funkcije je underlying query result set, a underlying query result set se evaluuira tek u SELECT fazi query-a, window funkcije smeju da se koriste samo u SELECT i ORDER BY clauseu querya.

Najcesce ces ih koristiti u SELECT-u. Ako treba da je referujes u ranijem logical query processing fazi poput WHERE, moraces koristiti table expression.

## Ranking window functions

Koristis ranking window functions kada rankujes svaki row with respect to others unutar window-a. T-SQL supports x4 ranking fn's: **ROW\_NUMBER, RANK, DENSE\_RANK, NTILE**.

```
SELECT orderid, custid, val,  
ROW_NUMBER() (ORDER BY val) as row_number,  
RANK() (ORDER BY val) as rank,  
DENSE_RANK() (ORDER BY val) as dense_rank,  
NTILE(10) OVER (ORDER BY val) as ntile  
FROM Sales.OrderValues  
ORDER BY val
```

orderid	custid	val	rownum	rank	dense_rank	ntile
10782	12	12.50	1	1	1	1
10807	27	18.40	2	2	2	1
10586	66	23.80	3	3	3	1
10767	76	28.00	4	4	4	1
10898	54	30.00	5	5	5	1
10900	88	33.75	6	6	6	1
10883	48	36.00	7	7	7	1
11051	41	36.00	8	7	7	1
10815	71	40.00	9	9	8	1
10674	38	45.00	10	10	9	1
...						
10691	63	10164.80	821	821	786	10
10540	63	10191.70	822	822	787	10
10479	65	10495.60	823	823	788	10
10897	37	10835.24	824	824	789	10
10817	39	10952.85	825	825	790	10
10417	73	11188.40	826	826	791	10
10889	65	11380.00	827	827	792	10
11030	71	12615.05	828	828	793	10
10981	34	15810.00	829	829	794	10
10865	63	16387.50	830	830	795	10

ROW\_NUMBER dodeljuje inkrementalni sekvencijalni integer na svaki red u query-u na osnovu window orderinga.

Nisam bas najbolje razumeo. RANK & DENSE\_RANK je za rangiranje valjda.

RANK reflects trenutni red koji ima manji ordering value od trenutnog reda. DENSE\_RANK reflects count distinct ordering values-a.

NTILE je kada zelis da povezes redove u rezultatu sa tileovima(equally sized groups of rows) dodeljivanjem tile numbera svakom redu. Specifikujes broj tilesova koje zelis i window ordering.

Sample query ima 830 redova i request je za 10 tiles.  $830/10 \Rightarrow 83$ . Window ordering se bazira na val column. To znaci da tih 83 reda sa najmanjim vrednostima ce biti dodeljeni vrednost 1. Oni sa najvisim vrednostima tile-u 10.

Svaka od ovih window funkcija podrzava i PARTITION BY koji da si stavio, onda umesto da se primenjuje na "sve redove", primenjivace se samo na subset redova koji je u partitionu za taj trenutni red koji se "izvrsava".

Window funkcije se logicki izvrsavaju kao deo SELECT-a, pre DISTINCT clausea. Zasto je to bitno? Trenutno OrderValues view ima 830 redova od kojih je 795 distinct vrednosti.

```
SELECT DISTINCT val, ROW_NUMBER() OVER(ORDER BY val) as rownum
FROM Sales.OrderValues;
```

orderid	custid	val	rownum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			

(830 rows affected)

Rezultat je 830 redova jer se ROW\_NUMBER procesira pre DISTINCT clause-a.

Alternativa bi bila da grupises sada redove na osnovu tog polja nad kojim radis ordering jer za svaki val ce se napraviti rownum.

```
SELECT val, ROW_NUMBER() OVER(ORDER BY val) as rownum
FROM Sales.OrderValues
GROUP BY val;
```

To ce da ti vrati 795 redova jer GROUP BY se izvrsava pre SELECT-a. Naravno, nema potrebe vise za DISTINCT.

## Offset window functions

Mozes koristiti offset window functions da bi vrati element iz reda koji je na odredjenom offsetu od trenutnog reda sa pocetka ili kraja window frame-a.

### **LAG, LEAD, FIRST\_VALUE, LAST\_VALUE**

LAG & LEAD supportuju window particije i windor-order clauses. Nema relevance sa window framingom. Ove fn koristis kako bi dobio element iz reda koji je u odredjenom offset-u od trenutnog reda unutar particije na osnovu orderinga.

**LAG** – gleda na vrednost pre trenutnog reda(looks before)

**LEAD** – gleda na vrednost posle trenutnog reda(looks ahead)

Prvi argument funkcija je obavezan i to je element koji zelis da vratis

Drugi argument je offset(1 if not specified)

Treci argument je default vrednost da vratis ako nema tog reda u requested offsetu(NULL default)

```

SELECT custid, orderid, val,
LAG(val) OVER( PARTITION BY custid ORDER BY orderdate, orderid) as prevval,
LEAD(val) OVER(PARTITION BY custid ORDER BY orderdate, orderid) as nextval
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid

```

custid	orderid	val	prevval	nextval
1	10643	814.50	NULL	878.00
1	10692	878.00	814.50	330.00
1	10702	330.00	878.00	845.80
1	10835	845.80	330.00	471.20
1	10952	471.20	845.80	933.50
1	11011	933.50	471.20	NULL
2	10308	88.80	NULL	479.75
2	10625	479.75	88.80	320.00

Koristis FIRST\_VALUE & LAST\_VALUE da vracas elemente koji su prvi/poslednji iz window frame-a, pa s toga podrzavaju window partition, window order & window frame clauses.

```

SELECT custid, orderid, val,
FIRST_VALUE(val) OVER( PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as firstval,
LAST_VALUE(val) OVER(PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as lastval
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid

```

custid	orderid	val	prevval	nextval
1	10643	814.50	NULL	878.00
1	10692	878.00	814.50	330.00
1	10702	330.00	878.00	845.80
1	10835	845.80	330.00	471.20
1	10952	471.20	845.80	933.50
1	11011	933.50	471.20	NULL
2	10308	88.80	NULL	479.75
2	10625	479.75	88.80	320.00

SQL Server 2022 je ubacio NULL Treatment clause sa offset window functions:

<function>(<expression>) [IGNORE NULLS | RESPECT NULLS] OVER(<specification>)

## Aggregate window functions

Aggregate window functions se koriste za agregiranje redova unutar definisanog "window-a". Podrzavaju window-partition, window-order & window-frame clauses.

```
SELECT orderid, custid, val,  
SUM(val) OVER(PARTITION BY custid) as customer_total  
SUM(val) OVER() as grand_total  
FROM Sales.OrderValues
```

U cemu je fora. Sum(val) OVER() exposeuje window nad svim redovima underlying query result set-a. Znaci svi redovi u Sales.OrderValues.

Kada radis PARTITION BY, onda dobijas window na samo one redove koji dele istu vrednost za tu neku kolonu(custid u ovom slucaju) naspram vrednosti reda koji invokeuje tu window funkciju.

orderid	custid	val	totalvalue	custtotalvalue
10643	1	814.50	1265793.22	4273.00
10692	1	878.00	1265793.22	4273.00
10702	1	330.00	1265793.22	4273.00
10835	1	845.80	1265793.22	4273.00
10952	1	471.20	1265793.22	4273.00
11011	1	933.50	1265793.22	4273.00
10926	2	514.40	1265793.22	1402.85

Evo recimo query koji predstavlja npr procenat trenutno vrednosti porudzbine naspram grand total i customer total-a

```
SELECT orderid, custid, val,  
       100. * val / SUM(val) OVER() as pctall,  
       100. * val / SUM(val) OVER(custid) as pctcust,  
FROM Sales.OrderValues;
```

orderid	custid	val	pctall	pctcust
10643	1	814.50	0.0643470029014691672941	19.0615492628130119354083
10692	1	878.00	0.0693636200705830925528	20.5476246197051252047741
10702	1	330.00	0.0260706089103558320528	7.7229113035338169904048
10835	1	845.80	0.0668197606556938265161	19.7940556985724315469225
10952	1	471.20	0.0372256694501808123130	11.0273812309852562602387
11011	1	933.50	0.0737482224782338461253	21.8464778843903580622513
10025	2	514.40	0.04062395401620810204181	26.5555074010011049149544

Ako dodamo npr window frame i window order

```

SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS runval
FROM Sales.EmpOrders;

```

empid	ordermonth	val	runval
1	2020-07-01	1614.88	1614.88
1	2020-08-01	5555.90	7170.78
1	2020-09-01	6651.00	13821.78
1	2020-10-01	3933.18	17754.96
1	2020-11-01	9562.65	27317.61
...			
2	2020-07-01	1176.00	1176.00

Kada zadajes neki window-frame, ti mozes doci i odrediti koliki frame zelis na deterministican nacin. To znaci da kazes "prethodna 2 reda i narednih 1 red"

ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING

## WINDOW Clause

Window clause ti dozvoljava da imenujes ceo window specification ili deo window spec-a u queryu i onda ga koristis u OVER clause-u window funkcija u tom queryu.

Poenta ovog clausea je da se skrati duzina querya. Dostupno je tek od sql server 2022 i higher.

Recimo da imas query koji koristi iste specifikacije nad aggregate funkcijama.

```

SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS runsum,
       MIN(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS runmin,
       MAX(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS runmax,
       AVG(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS runavg
FROM Sales.EmpOrders;

```

To se ponavlja x4 puta. Ono sto mozes jeste da kazes sledece

```
SELECT empid, ordermonth, val
SUM(val) OVER W AS runsum,
MIN(val) OVER W AS runmin,
MAX(val) OVER W AS runmax,
AVG(val) OVER W AS runavg
FROM Sales.EmpOrders
WINDOW W AS (PARTITION BY empid ORDER BY ordermonth ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW)
ORDER BY custid, orderdate, orderid
```

Mozes i extendovati taj window

```
SELECT custid, orderid, val,
FIRST_VALUE(val) OVER(
    PO ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS firstval,
LAST_VALUE(val) OVER(
    PO ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lastval
FROM Sales.OrderValues
WINDOW PO AS (PARTITION BY custid ORDER BY orderdate, orderid)
ORDER BY custid, orderdate, orderid;
```

Deli se ista particija i window-order, ali je drugaciji window-frame.

Za definisanje vise windowa

...

```
WINDOW P AS (PARTITION BY custid),
    PO AS (PARTITION BY ...),
    POF AS (PARTITION BY ...)
```

## Pivoting data

Pivoting data podrazumeva rotiranje podataka iz stanja “row”-a u stanje “column”-a.  
Postoji i mogucnost agregiranja podataka.

Pivoting se cesto izvrsava za svrhe data reportinga od strane prezentacionog sloja. Nekada to mozes i kroz db da odradis.

Recimo da imas sada ovaku tabelu.

```

USE TSQVL6;

DROP TABLE IF EXISTS dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid      INT          NOT NULL
        CONSTRAINT PK_Orders PRIMARY KEY,
    orderdate    DATE         NOT NULL,
    empid        INT          NOT NULL,
    custid       VARCHAR(5)  NOT NULL,
    qty          INT          NOT NULL
);

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
VALUES
    (30001, '20200802', 3, 'A', 10),
    (10001, '20201224', 2, 'A', 12),
    (10005, '20201224', 1, 'B', 20),
    (40001, '20210109', 2, 'A', 40),
    (10006, '20210118', 1, 'C', 14),
    (20001, '20210212', 2, 'B', 12),
    (40005, '20220212', 3, 'A', 10),
    (20002, '20220216', 1, 'C', 20),
    (30003, '20220418', 2, 'B', 15),
    (30004, '20200418', 3, 'C', 22),
    (30007, '20220907', 3, 'D', 30);

SELECT * FROM dbo.Orders;

```

Da moras sad da bratis total order quantity za svakog employee i customera uradis sledece

`SELECT empid, custid, SUM(qty) as sumqty`

`FROM dbo.Orders`

`GROUP BY empid, custid`

empid	custid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30

Recimo da moras da uradis total quantity po zaposlenom(redovi) na svakog kupca(kolone)

**TABLE 7-1** Pivoted view of total quantity per employee (on rows) and customer (on columns)

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Svaki pivoting request ukljucuje x3 logicke processing faze:

1. **Grouping** phase sa associated grouping ili na row elemente
2. **Spreading** phase sa associated spreading ili na cols elemente

### **3. Aggregating** faza sa associated aggregation elementima i aggregate funkcijama

---

Imagine a table of orders where each row includes an employee ID, a customer ID, and a quantity.

If you want to see how much each employee sold to each customer:

- Group the data by **employee ID**.(empid is the grouping element)
- Turn each **customer ID** into its own column (spread).(spreading quantities by customerid)
- Use **SUM(quantity)** to fill in the values. (you need to identify the aggregate function in the aggregation element(qty attribute) which is SUM in this case)

The chapter shows how to do this using:

- A **manual grouped query**, and
- The **PIVOT table operator**, which simplifies the process.

## Pivoting with a grouped query

Ova solucija je straightforward.

Grouping phase se radi sa GROUP BY

Spreading phase je sa SELECT clause uz CASE expression za svaki target column. Zbog case expressiona moras da znas spreading values u napred. Jer cemo spread po empid naspram customerida(a,b,c,d), onda ce to izgledati ovako:

CASE WHEN custid = 'A' THEN qty END

Taj izraz u sustini vraci quantity iz trenutnog reda samo kada je trenutni row za order customera A. Ukoliko ne specificiras ELSE, onda se vraca NULL.

Ako ne znas spreading values ahead of time, onda mozes queryovati te podatke i iz toga konstrusati query string.

Krajnji expression koji daje rezultat za customere:(dodaj jos c i d customere)

```
SELECT empid,  
SUM(CASE WHEN custid = 'A' THEN qty END) AS A,  
SUM(CASE WHEN custid = 'B' THEN qty END) AS B,  
FROM dbo.Orders GROUP BY empid;
```

## Pivoting with the pivot operator

Radi se sa PIVOT operatorom.

Kao table operator, PIVOT operates u contextu FROM clause-a. Izvrsava se nad source table ili table expression provided to it as its left input, pivots the data, and returns a result table.

Pivot operator ukljucuje istu logical processing fazu(grouping, spreading, aggregating) samo ne treba toliko linija koda da bi se postigla ista funkcionalnost.

Forma PIVOT operatora:

```
SELECT ...
FROM <input_table>
PIVOT (<agg_function>(<aggregation_element>)
       FOR <spreading_element> IN (<list_of_target_columns>)) AS <result_tbl_alias>
WHERE ...;
```

Spreading element ce biti custid

Aggregation element je qty

Aggregate function je SUM

list of target column names A, B, C, D

Ne moras eksplisitno specificirati grouping elemente pa samim tim GROUP BY otpada. U PIVOT operatoru, on ce sam skontati grouping elemente na osnovu eliminacije. Grouping elementi su svi atributi iz source table koji nisu bili specificirani kao spreading elementi ili aggregation elementi.

Zato moras voditi racuna da source table za PIVOT operator nema atribute druge sem grouping, spreading & aggregation elemenata!

To postizes koriscenjem table expressiona koji ce ukljuciti samo atribute koji su ti neophodni.

```
SELECT empid, A, B, C, D
FROM (SELECT empid, custid, qty FROM dbo.Orderes) as D
PIVOT(SUM(qty) FOR custid IN(A,B,C,D)) as P
```

## Unpivoting data

Unpivoting je tehnika koja rotira podatke iz stanja kolona u stanje rowova. U sustini ono po cemu si radio spreading, sada ce biti kao da si po tome radio grouping.

Uglavnom uključuje queryovanje pivoted stanja podataka i kreiranje iz svakog reda, vise result rows, svaki da razlicitom source column value.

Cesta upotreba je unpivot data koje si importovao iz spreadsheet u bazi zarad lakse manipulacije.

Kreiraj sada tabelu dbo.EmpCustOrders – predstavljace pivoted set.

```
USE TSQVL6;

DROP TABLE IF EXISTS dbo.EmpCustOrders;

CREATE TABLE dbo.EmpCustOrders
(
    empid INT NOT NULL
        CONSTRAINT PK_EmpCustOrders PRIMARY KEY,
    A VARCHAR(5) NULL,
    B VARCHAR(5) NULL,
    C VARCHAR(5) NULL,
    D VARCHAR(5) NULL
);

INSERT INTO dbo.EmpCustOrders(empid, A, B, C, D)
SELECT empid, A, B, C, D
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;

SELECT * FROM dbo.EmpCustOrders;
```

Here's the output of the query against *EmpCustOrders* showing its contents:

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Nerelevantni preseci(nema podataka) su takodje ukljuceni u ovom skupu i obelezeni sa NULL.

Ovaj problem se moze resiti na x2 nacina

- APPLY
- UNPIVOT

## Unpivoting with APPLY

Ovaj nacin uključuje:

1. Producing copies
2. Extracting values
3. Eliminating irrelevant rows

Prvi korak podrazumeva kreiranje kopije za svaki source row. Jedna za svaku kolonu koju nameravas da unpivotujes. To znaci da ces morati kreirati kopiju za svaku kolonu A,B,C & D. Cross join ti to resava izmedju EmpCustOrders tabelom i tabelom koja ima row za svaki customer.

Ako imas tabelu customera u bazi, onda mozes koristiti tu tabelu u cross join. Ako nemas, onda mozes kreirati virutelnu koriscenjem table-value constructor based on the VALUES clause.

```
SELECT * FROM dbo.EmpCustOrders
CROSS JOIN (VALUES('A'),('B'),('C'),('D')) AS C(custid)
```

VALUES clause definise skup x4 reda, svaki sa jednim customer ID value.

empid	A	B	C	D	custid
1	NULL	20	34	NULL	A
1	NULL	20	34	NULL	B
1	NULL	20	34	NULL	C
1	NULL	20	34	NULL	D
2	52	27	NULL	NULL	A
2	52	27	NULL	NULL	B
2	52	27	NULL	NULL	C
2	52	27	NULL	NULL	D
3	20	NULL	22	30	A
3	20	NULL	22	30	B
3	20	NULL	22	30	C
3	20	NULL	22	30	D

Drugi korak je extracting value iz jednog od izvornih customer quantity columns(A,B,C ili D) da bi se vratila jedna value colona(recimo da je nazivamo *qty*). Moraces extractovati vrednost iz column koja predstavlja trenutni custid value.

Ako je custid 'A', onda qty column treba da vrati vrednost iz kolone A. Ako je B, onda iz kolone B.

so on. To achieve this step, you might think you can simply add the *qty* column as a second column to each row in the table value constructor (the *VALUES* clause), like this:

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
CROSS JOIN (VALUES('A', A),('B', B),('C', C),('D', D)) AS C(custid, qty);
```

However, remember that a join treats its two inputs as a set; hence, there's no order between those inputs. You can't refer to the elements of either of the inputs when constructing the other. In our case, the table-value constructor on the right side of the join has references to the columns *A*, *B*, *C*, and *D* from the left side of the join (*EmpCustOrders*). Consequently, when you try to run this code, you get the following errors:

```
Msg 207, Level 16, State 1, Line 222
Invalid column name 'A'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'B'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'C'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'D'.
```

Iz tog razloga ide cross apply umesto cross join. Cross apply evaluira prvo levu stranu i onda primenjuje desnu stranu za svaki levi red i na taj nacin ce se moci pristupati elementima s leve strane u desnoj strani.

## CROSS JOIN => CROSS APPLY

This query runs successfully, returning the following output:

empid	custid	qty
1	A	NULL
1	B	20
1	C	34
1	D	NULL
2	A	52
2	B	27
2	C	NULL
2	D	NULL
3	A	20
3	B	NULL
3	C	22
3	D	30

Treci korak je sto imamo sada nullove za qty. Nema poente da ih drzimo. E sad to sto je CROSS APPLY izvrsen u toku FROM, znaci da njegove kolone su nam dostupne u WHERE i mozemo samo dodati filter da se te empty kolone ne uracunaju

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
CROSS APPLY (VALUES ('A', A), ('B',B), ('C',C), ('D',D)) AS C(custid, qty)
WHERE qty IS NOT NULL;
```

empid	custid	qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

## Unpivoting with the UNPIVOT operator

```
SELECT ...
FROM <input_table>
UNPIVOT(<values_column> FOR <names_column> IN (<source_columns>)) AS
<result_table_alias_> WHERE...
```

Unpivoting podrazumeva kreiranja x2 result columns iz bilo kog broja source kolona. Jedna koja drzi source column names kao stringove i druga koja drzi source column value. U

ovom primeru, moras unpivot source kolone ABC & D, producing the result names column custid i qty.

```
SELECT empid, custid, qty  
FROM dbo.EmpCustOrders  
UNPIVOT(qty FOR custid IN (A,B,C,D)) AS U;
```

UNPIVOT je implementiran kao table operator u kontekstu FROM clause-a. Izvrsava se nad source tabelom(ili table expression). Unutar zagrada unpitovt operatora, specificiras ime koje zelis da assignujes koloni koja ce drzati source column values i ime koje zelis da assignujes koloni koja ce drzati source column names(custid) i listu source column names(A,B,C & D). Posle zagrada, potom deklarisas alias table rezultu iz tog UNPIVOT table operatora.

UNPIVOT ce automatski ukloniti NULLS, nije optional kao unutar APPLY operatora.

## Grouping sets

U sustini query sa GROUP BY clause

```
SELECT SUM(qty) as sumqty FROM dbo.Orders; je takodje grouped set iako nema GROUP  
BY
```

Recimo da nad istom tabelom zelis da napravis uniju podataka nad razlicitim grupacijama.

```
SELECT empid, NULL, SUM(qty) AS sumqty  
FROM dbo.Orders  
GROUP BY empid  
  
UNION ALL  
  
SELECT NULL, custid, SUM(qty) AS sumqty  
FROM dbo.Orders  
GROUP BY custid  
  
UNION ALL  
  
SELECT NULL, NULL, SUM(qty) AS sumqty  
FROM dbo.Orders;
```

Prvi veliki problem ovde je performance query-a. Drugi problem je sto postoji vec native clause za ovakve stvari

## GROUPING SETS subclause

U sustini pojacava GROUP BY clause. Mozes ga koristiti da bi definisao visestruke grouping setove unutar istog querya.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS
(
    (empid, custid),
    (empid),
    (custid),
    ()
);

```

Poslednji grouping set => () je u sustini "nista" tj empty grouping set. Predstavlja grand total.

Ovo je logicki ekvivalentn onom gore queryu sa puno UNION-a, ali mnogo bolje se interno optimizuje i cistije je.

## CUBE subclause

Abbriated way to define multiple grouping sets.

U zagradi CUBE subclause-a ti postavis skup clanova razdvojene zarecom i uzmes sve moguce grouping skupove koji se mogu definisati na osnovu input membera

Npr CUBE(a, b, c) je ekvivalentno sledecem:

**GROUPING SETS( (a,b,c), (a,b), (a, c), (b,c), (a), (b), (c), () )**

U teoriji, skup svih podskupova elemenata koji se mogu kreirati iz odredjenog skupa. Naziva se power set.

```
SELECT empid, custid, SUM(qty) as sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Onaj gore query bi ovakav bio sa CUBE.

## ROLLUP subclause

Takodje nudi skracen(abbreviated) way to define multiple grouping sets.

ROLLUP ne produceuje sve moguce gruouping setove poput CUBE, ROLLUP zauzme hijerarhiju unutar input clanova i kreira samo grouping sets koji formiraju leading combinations of the input members.

**ROLLUP(a,b,c)** bi kreirao x4 grouping seta gde je hijerarhija a>b>c tj kao:

**GROUPING SETS( (a,b,c), (a,b), (a), () )**

Na primer da zelis da vratis total quantities za sve grouping sets koji mogu biti definisani baziranjem na one time hierarchy of order year, order month, order day.

```
GROUPING SETS(
    (YEAR(orderdate), MONTH(orderdate), DAY(orderdate)),
    (YEAR(orderdate), MONTH(orderdate)),
    (YEAR(orderdate)),
    () )
```

The logical equivalent that uses the *ROLLUP* subclause is much more concise:

```
ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
```

Here's the complete query you need to run:

```
SELECT
    YEAR(orderdate) AS orderyear,
    MONTH(orderdate) AS ordermonth,
    DAY(orderdate) AS orderday,
    SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

This query produces the following output:

orderyear	ordermonth	orderday	sumqty
2020	4	18	22
2020	4	NULL	22
2020	8	2	10
2020	8	NULL	10
2020	12	24	32
2020	12	NULL	32
2020	NULL	NULL	64
2021	1	9	40
2021	1	18	14
2021	1	NULL	54
2021	2	12	12
2021	2	NULL	12
2021	NULL	NULL	66
2022	2	12	10
2022	2	16	20
2022	2	NULL	30
2022	4	18	15
2022	4	NULL	15
2022	9	7	30
2022	9	NULL	30
2022	NULL	NULL	75
NULL	NULL	NULL	205

## GROUPING & GROUPING\_ID functions

Kada imas single query koji definise vise grouping setova, mozda bi morao treba da uvezes rezultate redova i grouping setova.

Dokle god su svi gruping elementi definisani sa NOT NULL, to je lagano.

```
SELECT empid, custid, SUM(qty) as sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid)
```

empid	custid	sumqty
2	A	52
3	A	20
NULL	A	72
1	B	20
2	B	27
NULL	B	47
1	C	34
3	C	22
NULL	C	56
3	D	30
NULL	D	30
NULL	NULL	205
1	NULL	54
2	NULL	79
3	NULL	72

NULL o kolonama ovde je u sustini placeholder koji kaze da ta kolona nije bila ukljucena u grupisanju skupa.

E sada, ukoliko grouping column dozvoljava nullove(custid, sumqty, empid), onda nemas bas precizan nacin da razlikujes redove u kojima zaista nema vrednosti ili kolona samo nije bila ukljucena u grupisanju skupa.

GROUPING fn je jedan od nacina da se to razresi. U sustini ona prihvata ime kolone i ukoliko je element “aggregate element” onda vrati 1, a vrati 0 ukoliko je u pitanju detail element(member of the current grouping set).

U sustini malo kontraintuitivno, ali it is what it is.

```

SELECT
    GROUPING(empid) AS grpemp,
    GROUPING(custid) AS grpcust,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);

```

This query returns the following output:

grpemp	grpcust	empid	custid	sumqty
0	0	2	A	52
0	0	3	A	20
1	0	NULL	A	72
0	0	1	B	20
0	0	2	B	27
1	0	NULL	B	47
0	0	1	C	34
0	0	3	C	22
1	0	NULL	C	56
0	0	3	D	30
1	0	NULL	D	30
1	1	NULL	NULL	205
0	1	1	NULL	54
0	1	2	NULL	79
0	1	3	NULL	72

I sada ne moras vise da se oslanjas na null da bi povezao vezu izmedju result rows i grouping sets.

GROUPING\_ID funkcija moze da ga malo pojednostavi. U sustini das funkciju sa svim elementima koji su ukljeceni u grouping set kao input : GROUPING\_ID(a, b, c, d) i fn vrati integer bitmap u kom svaki bit predstavlja razliciti input element.

GROUPING\_ID function above, the grouping set (a, b, c, d) is represented by the integer 0 ( $0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1$ ). The grouping set (a, c) is represented by the integer 5 ( $0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$ ), and so on

```
SELECT
    GROUPING_ID(empid, custid) AS groupingset,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

This query produces the following output:

groupingset	empid	custid	sumqty
0	2	A	52
0	3	A	20
2	NULL	A	72
0	1	B	20
0	2	B	27
2	NULL	B	47
0	1	C	34
0	3	C	22
2	NULL	C	56
0	3	D	30
2	NULL	D	30
3	NULL	NULL	205
1	1	NULL	54
1	2	NULL	79
1	3	NULL	72

integer 0 predstavlja grouping set (empid custid) (00)

Integer 1 predstavlja grouping set (empid) (01)

Integer 2 predstavlja grouping set (custid) (10)

Integer 3 predstavlja grouping set () (11)

## Time series

Time series data su podaci koji predstavljaju "seriju" dogadjaja(series of events) ili mera(measurements) koje su uzete na odredjenom vremenskom intervalu.

Primer: Podaci o temperaturi i vlaznosti na svaki sat. Time series data analysis cesto uključuje organizovanje podataka po grupama(**buckets**) i onda agregiranje istih po bucketu.

### Sample data:

Recimo imamo senzor temperature i vlaznosti koji hvata podatke intervalno. Podaci ce se cuvati u x2 tabele – Sensors(podaci o senzorima) & SensorMeasurements(ono sto senzori iscitaju)

```
DROP TABLE IF EXISTS dbo.SensorMeasurements, dbo.Sensors;

CREATE TABLE dbo.Sensors
(
    sensorid      INT          NOT NULL
        CONSTRAINT PK_Sensors PRIMARY KEY,
    description VARCHAR(50) NOT NULL
);

INSERT INTO dbo.Sensors(sensorid, description)
VALUES
    (1, 'Restaurant Fancy Schmancy beer fridge'),
    (2, 'Restaurant Fancy Schmancy wine cellar');

CREATE TABLE dbo.SensorMeasurements
(
    sensorid      INT NOT NULL
        CONSTRAINT FK_SensorMeasurements_Sensors REFERENCES dbo.Sensors,
    ts            DATETIME2(0) NOT NULL,
    temperature  NUMERIC(5, 2) NOT NULL, -- Fahrenheit
    humidity     NUMERIC(5, 2) NOT NULL, -- percent
    CONSTRAINT PK_SensorMeasurements PRIMARY KEY(sensorid, ts)
);

INSERT INTO dbo.SensorMeasurements(sensorid, ts, temperature, humidity)
VALUES
    (1, '20220609 06:00:03', 39.16, 86.28),
    (1, '20220609 09:59:57', 39.72, 83.44),
    (1, '20220609 13:59:59', 38.93, 84.33),
    (1, '20220609 18:00:00', 39.42, 79.66),
```

Stranica 263. su gde se ovi podaci nalaze.

### DATE\_BUCKET Function

#### DATE\_BUCKET(datepart, bucketwidth, ts[,origin])

Azure sql edge, MSSQL 2022 & Azure SQL database

Poenta funkcije je da vrati **pocetnu tacku time bucket-a** koji sadrzi input timestamp.

Rezultat funkcije moze se koristiti kao identifier of the containing bucket i gruping element u queryu.

Kako bi identifikovao containing time bucket za neke input timestamp(ts), prvo moras razmisiliti o “strelji vremena” koja se deli u time buckets. Znaci neophodna ti je pocetna tacka bucketa.

Ta pocetna tacka se definise sa *origin* inputom koji moze biti bilo koji date and time data type i opcion je. Ako nije specified, onda je default 1. jan 1900.

*datepart* specificiras date and time part koji zelis da koristis poput year/month/day/hour/second i sl.

*bucketworkwidth* specifyuje bucket width u smislu specificiranog dela(part). Npr ako kazes *datepart: hour & bucketwidth: 12* ti definises 12 hour buckets. Input *ts* je input timestamp za koji hoces da vratis start time za containing bucket. On moze biti bilo koji date and time i trebao bi da se matchuje sa inputom origina(po pitanju tipa). Function's return type je isti kao i tip inputa za *ts*.

```
DECLARE
    @ts          AS DATETIME2(0) = '20220102 12:00:03',
    @bucketwidth AS INT = 12,
    @origin      AS DATETIME2(0) = '20220101 00:05:00';

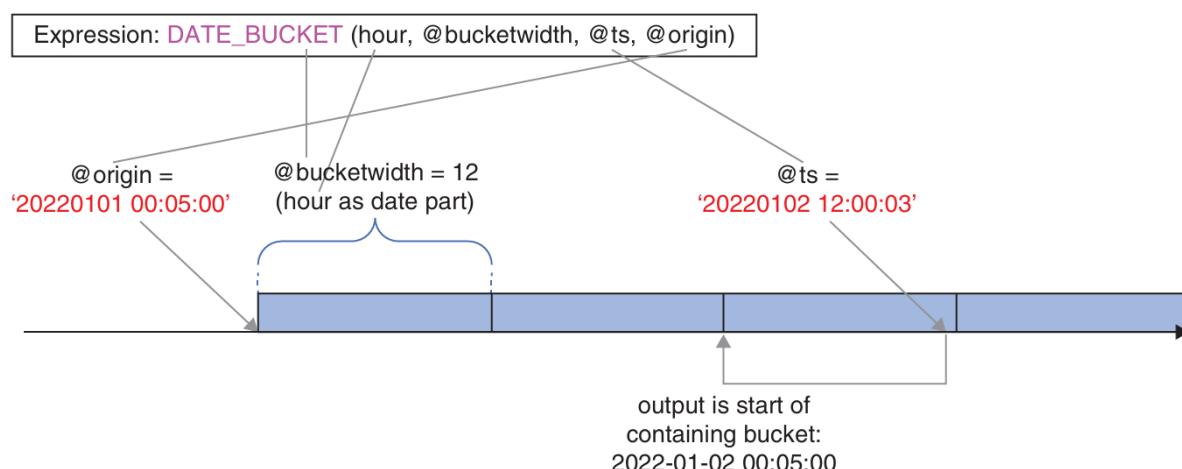
SELECT DATE_BUCKET(hour, @bucketwidth, @ts, @origin);
```

Here you define 12-hour buckets, starting with 2022-01-01 00:05:00 as the origin point, and request the starting point of the bucket containing the timestamp 2022-01-02 12:00:03.

This code generates the following output:

```
-----
2022-01-02 00:05:00
```

Figure 7-1 can help you understand visually how the function works and the logic leading to the function's result.



Prva x4 12h bucketa imaju sledeca start vremena:

2022 01 01 00 05 00 -> 2022 01 01 12 05 00 -> 2022 01 02 00 05 00 -> 2022 01 02 12 05 00

Input timestamp 2022-01-02 12:00:03 je u x3 bucketu i stoga funkcija vraca vrednost: 2022-01-02 00:05:00 koja je start tog containing bucketa.

## Custom computation of start of containing bucket

stranica 268 ako si zainteresovan.

## Applying bucket logic to sample data

Tipicno zelis da bucketizeujes time series stored u tabeli poput *SensorMeasurements*. Recimo zelis da saljes upite nad tebelom i za svako citanje, computeujes respective start of the bucket.

Prvo definises 12 hour buckets starting ad midnight i onda mozes koristiti ponoc bilo kog datuma kao origin.

DECLARE

```
@bucketwidth as INT = 12  
@origin as DATETIME2(0) = '19000101 00:00:00';
```

```
SELECT sensorid, ts, DATE_BUCKET(hour, @bucketwidth, ts, @origin) as bucketstart  
FROM dbo.SensorMeasurements
```

input ovoj funkciji sada je kolona "ts" iz tabele.

Both queries generate the following output:

sensorid	ts	bucketstart
1	2022-06-09 06:00:03	2022-06-09 00:00:00
1	2022-06-09 09:59:57	2022-06-09 00:00:00
1	2022-06-09 13:59:59	2022-06-09 12:00:00
1	2022-06-09 18:00:00	2022-06-09 12:00:00
1	2022-06-09 22:00:01	2022-06-09 12:00:00
1	2022-06-10 01:59:57	2022-06-10 00:00:00
1	2022-06-10 05:59:59	2022-06-10 00:00:00
1	2022-06-10 09:59:58	2022-06-10 00:00:00
1	2022-06-10 14:00:03	2022-06-10 12:00:00
1	2022-06-10 17:59:59	2022-06-10 12:00:00
1	2022-06-10 21:59:57	2022-06-10 12:00:00
1	2022-06-11 01:59:58	2022-06-11 00:00:00
1	2022-06-11 06:00:03	2022-06-11 00:00:00
1	2022-06-11 10:00:00	2022-06-11 00:00:00
1	2022-06-11 14:00:02	2022-06-11 12:00:00
1	2022-06-11 18:00:02	2022-06-11 12:00:00
1	2022-06-11 21:59:59	2022-06-11 12:00:00
-	.....	.....

I sada ono sto cesto zelis da uradis jeste da grupises podatke po bucketu, a ne samo da ih podelis i onda vrsis aggregate funkcije nad grupama.

Npr napravis onaj gore da je CTE uz WITH i onda

WITH c as (...)

SELECT sensorid, bucketstart,

DATEADD(hour, @bucketwidth, bucketstart) as bucketend,

MIN(temperature) as mintemp,

MAX(temperature) as maxtemp,

AVG(temperature) as avgtemp

FROM c

GROUP BY sensorid, bucketstart

ORDER BY sensorid, bucketstart

sensorid	bucketstart	bucketend	mintemp	maxtemp	avgtemp
1	2022-06-09 00:00:00	2022-06-09 12:00:00	39.16	39.72	39.440000
1	2022-06-09 12:00:00	2022-06-10 00:00:00	38.93	40.08	39.476666
1	2022-06-10 00:00:00	2022-06-10 12:00:00	40.03	41.26	40.726666
1	2022-06-10 12:00:00	2022-06-11 00:00:00	39.32	41.23	40.580000
1	2022-06-11 00:00:00	2022-06-11 12:00:00	39.20	41.14	40.406666
1	2022-06-11 12:00:00	2022-06-12 00:00:00	39.41	41.12	40.400000
1	2022-06-12 00:00:00	2022-06-12 12:00:00	39.23	41.40	40.593333
1	2022-06-12 12:00:00	2022-06-13 00:00:00	40.11	41.20	40.726666

Dobijes ovako nesto.

## Gap filling

Ovo gore je neki happy scenario gde manje vise vecina podataka je popunjena. Recimo da je jedan senzor bio offline i nema podataka/citanja, onda ces u sustini morati nekako popuniti te "prazne buckete".

Npr zelis podatke o vremenu od 9. juna 2022 do 15. juna 2022 po 12h intervalima.

Agregiranjem "praznih bucketa" gde nije bilo podataka ce rezultat biti NULL.

Gap-filling je cesto hangled organizovanjem tabele koja drzi sve tacke koje cemo posmatrati(podatke). U nasem slucaju to bi znacilo sve moguce bucket start times u nedelji.

Potom bi apply left outer join izmedju te tabele i bucketized podataka sa missing bucketima. Na taj nacin dobijas sve bucket informacije ukljucujuci i missing buckets.

Referentna tabela moze biti base tabela koja se rucno kreira ili table expression.

```
DECLARE  
    @bucketwidth AS INT = 12,  
    @startperiod AS DATETIME2(0) = '20220609 00:00:00',  
    @endperiod    AS DATETIME2(0) = '20220615 12:00:00';  
  
SELECT DATEADD(hour, value * @bucketwidth, @startperiod) AS ts  
FROM GENERATE_SERIES(0, DATEDIFF(hour, @startperiod, @endperiod) / @bucketwidth) AS N;
```

This code generates the following output:

```
ts  
-----  
2022-06-09 00:00:00  
2022-06-09 12:00:00  
2022-06-10 00:00:00  
2022-06-10 12:00:00  
2022-06-11 00:00:00  
2022-06-11 12:00:00  
2022-06-12 00:00:00
```

CHAPTER 7 T-SQL for data analysis

```
2022-06-12 12:00:00  
2022-06-13 00:00:00  
2022-06-13 12:00:00  
2022-06-14 00:00:00  
2022-06-14 12:00:00  
2022-06-15 00:00:00  
2022-06-15 12:00:00
```

(14 rows affected)

Ukratko, bata je kreirao time series od pocetnog do krajnjeg datuma.

Mozes sad koristiti multi-CTE sa CTE za svaki korak.

Na stranici 277 imas bas objasnjenje o tome.

## Data Modification

DML – Data Modification Language

Ukljucuje data modification & data retrieval.

Statements: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, MERGE

### Inserting data

Imas opcije: INSERT VALUES, INSERT SELECT, INSERT EXEC, SELECT INTO & BULK INSERT

## INSERT VALUES Statement

Koristis da insertujes redove u tabelu na specificne vrednosti.

**INSERT INTO table\_name[, (table\_columns)]**

**VALUES (row), (row), (row)**

Ukoliko ne specificiras vrednost za neku kolonu, sql ce koristiti default vrednost te kolone koja je NULL ukoliko nije definisano. Ukoliko null nije dozvoljen, onda ce failovati pa je ok.

Ukoliko koristis INSERT VALUES kako bi insertovao vise row-ova(kao ovaj gore), onda se taj deo insertovanja u pozadini obavlja unutar transakcije tako da ako jedna padne, sve padaju.

Taj enhanced VALUES ima vise upotreba. Na primer mozes ga koristiti i kao table-value construct za derived table.

**SELECT \* FROM (VALUES (row), (row), (row)) as O(orderid, orderdate, empid, custid)**

orderid	orderdate	empid	custid
10003	20220213	4	B
10004	20220214	1	A
10005	20220213	1	C
10006	20220215	3	C

## INSERT SELECT Statement

Insert select ubacuje skup redova koje vraca SELECT query

Umesto VALUES u sustini koristis SELECT.

**INSERT INTO dbo.Orders**

**SELECT orderid, orderdate, empid, custid**

**FROM Sales.Orders**

**WHERE shipcountry = 'UK'**

Sva pravila za default/null/fail vaze i ovde. INSERT SELECT se takodje izvrsava u transakciji. U ovom slucaju ukoliko iskoristis neku od sistemskih funkcija(npr SYSDATETIME), ta funkcija ce se invoke samo jednom i bice deljena za sve row-ove. Nece se izvrsiti svaki put za svaki red. Jedini exception je ako koristis neki globally unique identified generator koji se invoke svaki put drugaciji.

## INSERT EXEC Statement

Koristis da insertujes result set koji ti se vraca iz stored procedure ili dynamic sql batch u tabelu.

```
CREATE OR ALTER PROC Sales.GetOrders
@country as NVARCHAR(40)
AS
SELECT orderid, orderdate, empid, custid FROM Sales.Orders
WHERE shipcountry = @country;
GO

INSERT INTO dbo.Companies
EXEC Sales.GetOrders @country = 'France'
```

### **SELECT INTO Statement**

SELECT INTO Statement je non-standard T-SQL koja kreira target table & populateuje je sa rezultatima iz query-a.

By non standard, označava da nije deo ISO & ANSI SQL standarda.

Ne možes koristiti taj statement da bi insertovao podatke u postojeću tabelu.

**DROP TABLE IF EXISTS dbo.Orders;**

```
SELECT orderid, orderdate, empid, custid
INTO dbo.Orders
FROM Sales.Orders;
```

Struktura tabele i podaci se baziraju na source table. SELECT INTO kopira iz source table osnovnu strukturu(column names, types, nullability & identity property) & podatke. Ne kopira jedino source constraintove, indexe, triggere, column properties & permissions. To moras rucno ako ti treba.

Uz EXCEPT statement možes raditi dodatno filtriranje. Na primer

**DROP TABLE IF EXISTS dbo.Locations;**

```
SELECT country, region, city
INTO dbo.Locations
FROM Sales.Customers
EXCEPT
SELECT country, region, city
FROM HR.Employees
```

Ono što će se desiti jeste da će se prvo selectovati svi distinct Sales.Customers. Potom kroz EXCEPT query će se sva preklapanja sa tim filterom "otkloniti" i ostane samo sa rows

iz prvog query-a koja se ne pojavljuju u drugom queryju. Znaci samo Customer locations bez Employee locations.

## BULK Insert Statement

Koristis BULK Insert kako bi insertovao u postojeću tabelu podataka koja dolazi iz nekog file-a.

Specificiras target table, source file & options.

```
BULK INSERT dbo.Orders FROM 'c:\temp\orders.txt'
```

```
WITH (
  DATAFILETYPE = 'char',
  FIELDTERMINATOR = ',',
  ROWTERMINATOR = '\n'
);
```

Ovaj kod bulk insertuje sadržaj orders.txt u tabelu dbo.orders, specificirajući da data file type je char, field terminator je comma & row terminator je newline character. Znaci ovako nekako bi trebao file da izgleda:

```
vrednost, vrednost, vrednost  
vrednost, vrednost, vrednost  
vrednost, vrednost, vrednost
```

## The identity property and the sequence object

SQL Server supports x2 built in solution da automatski generise **numeric keys**

Identity column property & Sequence object.

Identity property je dobar za jedne scenarie, ali ima mnogo ogranicenja. Sequence object je tu da ih razresi.

Primer: kreiras primary key koji automatski generise vrednosti.

## Identity

Identity je u sustini samo standard column property. Može se definisati uz bilo koji numeric tip koji nema ostatak(no fraction). Opciono možes definisati increment(step value) i prvi value, ali ako to ne uradis default vrednost je 1.

U sustini koristis ovo kako bi generisao “surrogate keys” kojima upravlja sistem, a ne aplikacija.

```
CREATE TABLE dbo.T1
(
keycol INT NOT NULL IDENTITY(1,1)
    CONSTRAINT PK_T1 PRIMARY KEY,
datacol VARCHAR(10) NOT NULL
    CONSTRAINT CHK_T1_datacol CHECK(datacol LIKE '[ABCD]%' )
);
```

Sada u svojim insert statementima moras kompletno ignorisati identity kolonu(keycol u ovom slucaju)

U sustini SQL Server ce generisati novi identity na osnovu current identity vrednosti u tabeli i increment. Ako treba da obtainujes novogenerisani identity value kako bi insertovao child rows u referencing table – mozes query jednu od dve funkcije

**@@identity & SCOPE\_IDENTITY**

**@@identity** – vraca poslednji identity value generated od strane sesije bez obzira na scope. Npr stored procedura i INSERT koji ubacuju novi value kroz identity ce imati razlicite scopeove.

**SCOPE\_IDENTITY** – vraca poslednji identity value generisan od strane trenutnog scope-a(npr ista procedura). U retkim slucajevima kada ti nije stalo do scopea, svakako bi trebao koristiti SCOPE\_IDENTITY fn.

```
DECLARE @new_key AS INT;
INSERT INTO dbo.T1(datacol) VALUES('AAAAAA');
SET @new_key = SCOPE_IDENTITY();
SELECT @new_key AS new_key;
```

Dobijes vrednost “4” sto je u sustini poslednji novododani identity.

Oba @@identity & SCOPE\_IDENTITY() vracaju poslednje dodati identity UNUTAR SESIJE. To znaci da ukoliko neka druga sesija doda novi identity, on to nece prepoznati.

Ukoliko zelis da znas trenutni identity value u tabeli(poslednji identity kreiran u tabeli) bez obzira na sesiju, trebao bi koristiti **IDENT\_CURRENT** i dadas table name kao input.

```
SELECT
SCOPE_CURRENT() as 'scope curr',
@@identity as [@@identity],
IDENT_CURRENT('dbo.T1') as 'iden_cur';
```

Dobijes sledece:

SCOPE_IDENTITY	@@identity	IDENT_CURRENT
NULL	NULL	4

Ovi su vratili NULL jer je batica otvorio novu sesiju i izvrsio te funkcije.

Takodje jedan jako questionable design choice je da ukoliko uradis INSERT u neku tabelu koja ima IDENTITY property, insert failuje, taj interni identity ce se SVAKAKO povecati za step tj "obradice se" i nece se revertovati.

Tako da npr da je input sa 4 failovao, poslednji insertovani je 3, onda na sledecem insertu koji je uspesan dobices 5 kao ID.

SQL Server koristi performance cache feature za identity property inace sto moze rezultovati u prazninama izmedju kljuceva kada postoji unclean termination u SQL Server process-u npr ako padne struja.

Jos jedna mana IDENTITY kolone je sto ne mozes da je promenis. Znaci ne mozes da je otkacis sa kolone jednom kad je postavi, a ne mozes ni da je dadas na vec postojeći kolonu.

Uz SQL Server mozes specificirati svoju explicit vrednost za identity kolonu kada insertujes redove dokle god enableujes opcije **IDENTITY\_INSERT** nad tabelom. Ono sto ne mozes je da updateujes identity jednom kada ga postavis.

```
SET IDENTITY_INSERT dbo.T1 ON;  
INSERT...  
SET IDENTITY_INSERT dbo.T1 OFF;
```

Interesantno je da kad iskljucis IDENTITY INSERT opciju, sql server promeni current identity value u tabeli jedino ako explicit value koji si provideovao je veca od trenutno generisane vrednosti.

To znaci da ako je poslednji najveci identity "6", ti si sad ubacio "4" kroz enableovanje identity\_inserta, onda ce sledeci i dalje biti "7".

Takodje IDENTITY ne enableuje uniqueness. Moras to postaviti rucno nad celom kolonom ili da je defineujes kao primary key. IDENTITY ne mora nuzno samo na PK kolone.

## Sequence

T-SQL podrzava standardni sequence object kao alternativni key-generation mehanizam za identitet. Mnogo je fleksibilniji.

- **Jedna od prednosti** je sto sequence **nije vezan za konkretnu kolonu**. To je nezavisni objekat u bazi. Kad treba da generises novu vrednost, invokeujes funkciju nad objektom i koristis vracenu vrednost. Scenario – mozes koristiti jedan sequence object koji ce ti pomoci da odrzavas kljuceve koji se nece conflictovati izmedju razlicitih tabela.

```
CREATE SEQUENCE <sequence_name> AS <type>
```

Default tip je BIGINT ako ne specificiras <type>. Tip moze biti bilo koji **numeric type with a scale of zero**.

- **Druga prednost** je postojanje MINVALUE <val> & MAXVALUE <val> specifikacija unutar tipa. Ako ih ne naglasis, koristice se minvalue & maxvalue dodeljenog tipa.

**Za razliku od identity, sequence object podrzava cycling.** To znaci da kada dodjes do maxvalue, on nastavlja na sledeci minimum value. **NO CYCLE** je default ponasanje pa ako zelis da **omogucis cikliranje** onda moras postaviti **CYCLE** option.

Sequence object **podrzava kao i identity** podesavanje pocetne vrednosti (**START WITH <val>**) i step increment (**INCREMENT BY <step>**). Ako ne postavis, onda ce start biti na MINVALUE, a step ce biti 1.

Npr recimo da zelis sekvencu koja ce ti pomoci pri generisanju order IDs. Zelis da se ciklira i kreće od 1.

```
CREATE CYCLE dbo.SeqOrderIDs AS INT  
MINVALUE 1  
CYCLE;
```

Sequence takodje **podrzava caching** (CACHE <val> | NO CACHE) koji govori SQL Serveru koliko cesto da upisuje recoverable value to disk. Npr CACHE 10000 ce mu reci "upisi value u disk na svakih 10000 requests, a izmedju desk writes ces cuvati trenutnu vrednost i koliko ih je ostalo u samoj memoriji.

Naravno, sto manje pises u disk to bolji performans, ali manja pouzdanost. Default je 50.

**MOZES** promeniti neku od svojstava sequence-a koriscenjem **ALTER SEQUENCE** (MINVAL <val>, MAXVAL <val>, RESTART WITH <val>, INCREMENT BY <val>, CYCLE | NO CYCLE, CACHE <val> | NO CACHE).

```
ALTER SEQUENCE dbo.SeqOrderIDs
NO CYCLE;
```

Da generises novu sequence vrednost uradis:

```
SELECT NEXT VALUE FOR dbo.SeqOrderIDs;
```

I na taj nacin mozes da storeujes vrednost sekvence u variable i koristis variable ili jednostavno inline da dodjes i napises sekencu.

Da dobijes info o twojim sekvencama, radis query na **sys.sequences**. Npr trenutni sequence value u SeqOrderIDs:

```
SELECT current_value
FROM sys.sequences
WHERE OBJECT_ID = OBJECT_ID('dbo.SeqOrderIDs');
```

```
current_value
-----
5
```

SQL Server extends podrsku za sequence. Jedna od ekstenzija ti omogucuje da kontrolises redosled assigned sequence values u multirow insertu koriscenjem OVER clause-a.

Npr:

```
INSERT INTO dbo.T1(keycol, datacol)
SELECT NEXT VALUE FOR dbo.SeqOrderIDs OVER(ORDER BY hiredate), LEFT(firstname, 1)
+ LEFT(lastname, 1) FROM HR.Employees;
```

```
SELECT * FROM dbo.T1
```

**Jos jedna ekstenzija jeste koriscenje NEXT VALUE FOR funkciju u default constraint!!!**

```
ALTER TABLE dbo.T1
ADD CONSTRAINT DFT_T1_keycol
DEFAULT (NEXT VALUE FOR dbo.SeqOrderIDs)
FOR keycol;
```

Sad mozes insertovati redove u tabelu bez da specificno deklarisas vrednost za keycol. Pa cak mozes i remove taj constraint za razliku od IDENTITY.

Postoji i ekstenzija za assignovanje sekvence vrednosti odjednom koriscenjem stored procedure **sp\_sequence\_get\_range**. Ideja je da ako aplikacija treba da assignuje range sequence vrednosti, efikasnije je updateovati sekvencu samo jednom, incrementovati je za size of the range. Primer za pozivanje proc kako bi dobio 1\_000\_000 vrednosti

```
DECLARE @first as SQL_VARIANT;  
  
EXEC sys.sp_sequence_get_range  
    @sequence_name = 'dbo.SeqOrderIDs',  
    @range_size = 1000000,  
    @range_first_value = @first OUTPUT;  
  
SELECT @first;
```

SQL\_VARIANT je generic data type koji moze da drzi u sebi razlicite data typeove. Kao dynamic neki. sp\_sequence\_get\_range koristi taj tip za nekoliko njihovih parametara kao i @range\_first\_value.

- **Sekvenca ne garantuje da nenes imati gaps. Ako se nova sequence value generisala u transakciji koja je failovala ili se rollbackovala, sequence change nije undone isto kao i u identity.**

## Deleting data

T-SQL nudi DELETE & TRUNCATE za brisanje podataka.

### DELETE Statement

Standard statement koji se koristi da brise podatke iz tabele na osnovu optional filter predicate.

```
DELETE FROM <table_name>  
WHERE orderdate < '20210101'
```

Reci ce (152 rows affected) npr, ali mozes supress rezultat sa “NOCOUNT”

DELETE statement moze da bude skupa kada brise veliki broj redova jer je fully logged operacija.

## TRUNCATE Statement

TRUNCATE brise sve podatke iz tabele i ne podrzava filtere.

### TRUNCATE TABLE dbo.T1;

Prednost truncate-a je da je minimalno logged pa ce mnogo brze da se izvrsi. Ako imas npr tabelu sa milionima redova i trebas da ih ocistis skroz, DELETE ce zahtevati minute dok TRUNCATE sekunde.

Oba DELETE & TRUNCATE su transakcioni tako da ako nesto failuje u transakciji pri brisanju, mogu se oba revert.

Kada tabela ima IDENTITY column, truncate & delete se i tu razlikuju. DELETE ne resetuje identity value dok TRUNCATE resetuje. SQL Standard definise identity column restart option za TRUNCATE, ali to "kao opcija" nije implementirano. Znaci reset se "mora" desiti.

TRUNCATE dozvoljava truncateovanje individualnih particija.

```
TRUNCATE TABLE dbo.T1 WITH ( PARTITIONS(1, 3, 5, 7 TO 10) );
```

## DELETE based on a join

T-sql podrzava nonstandard DELETE syntax na osnovu joina. Join sluzi u filtering svrhe i daje ti access na atribute related rowsa iz joined tabele.

U sustini mozes da brises redove iz jedne tabele na osnovu vrednosti u drugoj tabeli.

### DELETE FROM O

```
FROM dbo.Orders AS O
```

```
INNER JOIN dbo.Customers AS C ON O.custid = C.custid
```

```
WHERE C.country = 'USA';
```

Kako je ovo nonstandard, mozes i na drugi nacin napisati sl kod:

```
DELETE FROM dbo.Orders
```

```
WHERE EXISTS (SELECT * FROM dboCustomers as c WHERE Orders.custid = c.custid AND c.country = 'USA')
```

No perf. diff. ofc.

## Updating data

Supports standard **UPDATE** statement koji mozes koristiti da updateujes redove u tabeli.

Takodje postoje nonstandard forms of the UPDATE statement sa joins & variables.

## **UPDATE Statement**

Standardni statement koji mozes koristiti da updatejes podskup redova u tabeli.

```
UPDATE <table_name>
SET <column_1> = <new_value>, <column_2> = <new_value>
WHERE <filter>
```

**UPDATE** statement je takodje all at once operacija.

To znaci:

```
UPDATE dbo.T1
SET col1 = col1 + 10, col2 = col1 + 10;
```

Assignmenti ce se desiti all at once, odjednom. Oba assignmenta ce koristiti vrednost col1 u tom momentu pre updatea. Recimo da je col1 bio 100. To znaci da ce on sada biti 110. col2 ce takodje biti 110. Nece biti 120.

To takodje znaci da:

```
UPDATE dbo.T1
SET col1 = col2, col2 = col1;
```

Ovaj izraz nece zahtevati neku bucket vrednost. Zbog all at once.

## **UPDATE based on a join**

Slicno kao i DELETE, UPDATE takodje podrzava nonstandard formu koja se bazira na joinu.

UPDATE keyword in this scenario should be followed by the alias of the table that is the target of the update. You can't update more than one table in the same statement, followed by the set clause with column assignments. It's like a select, only the top part is an update.

```
UPDATE OD
SET discount += 0.05
FROM dbo.OrderDetails as OD
INNER JOIN dbo.Orders as O ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

U smislu logical processinga, kreće se od FROM, dodje do WHERE i onda tek UPDATE. Kao SELECT, UPDATE je poslednji koji se izvršava.

Standard code za ovu nonstandard rezoluciju:

```
UPDATE dbo.OrderDetails
SET discount += 0.05
WHERE EXISTS (SELECT * FROM dbo.Orders as O WHERE o.orderid =
OrderDetails.orderid AND o.custid = 1);
```

Slucajevi u kojima imaju prednost ovi fensi T-SQL varijante su sledeci npr:

```
UPDATE T1
SET col1 = T2.col1, col2 = T2.col2, col3 = T2.col3
FROM dbo.T1
JOIN dbo.T2 ON T1.keycol = T2.keycol
WHERE T2.col4 = 'ABC';
```

Da npr moramo na “dirty way”

```
UPDATE dbo.T1
SET col1 = (SELECT col1 FROM dbo.T2 WHERE T2.keycol = T1. keycol),
col2 = (SELECT col2 FROM dbo.T2 WHERE T2.keycol = T1. keycol),
col3 = (SELECT col3 FROM dbo.T2 WHERE T2.keycol = T1. keycol)
WHERE EXISTS (SELECT * FROM dbo.T2 WHERE T2.keycol = T1.keycol AND T2.col4 = 'ABC')
```

Za svaki row u T1, on ce ulaziti u T2 i izvaditi vrednosti kolona u redovima gde je filter “true” tj exists veza za T2.col4 = ‘ABC’.

Ova verzija je manje efikasna jer cemo ulaziti x4 puta u join. U ovoj T-SQL nonstandard verziji cemo ulaziti samo jednom. Mnogo efikasnije.

## Assignment UPDATE

T-SQL supports sintaksu koja updateuje podatke u tabeli, a u isto vreme i dodaje vrednosti u promenljive. Sintaksa je takva da te “sacuva” od separate UPDATE & SELECT statementa.

Use case bi bio odrzavanje custom sequence/autonumbering mehanizma kad nemas identity column property & sequence object i kada ne zelis da imas “praznnine” izmedju vrednosti.

Korak 1: Kreiraj tabelu koja ce da sadrzi key->val vrednosti za tvoje “sekvence”

```
CREATE TABLE MySequences(
id VARCHAR(10) NOT NULL PRIMARY KEY,
val INT NOT NULL);
```

```
INSERT INTO MySequences('SEQ1',0);
```

I sada svaki put kada se odradi neki update, ti ces inkrementirati tu custom sekvencu za +1

```
DECLARE @nextval AS INT;  
  
UPDATE dbo.MySequences SET @nextvale = val+=1  
WHERE id = 'SEQ1';  
  
SELECT @nextval;
```

Sta se desava. Imas local variable @nextval. Potom je tu UPDATE syntax a incrementovanje kolone za +1 i assignovanje nove vrednosti u variable. Kod potom predstavlja value u variable. First "val" je set na "val + 1" i onda "nextval" preuzima tu vrednost.

To znaci da "nextval == MySequences.val"

Mozes cak to izbaciti u funkciju i vratiti vrednost "nextval"-a i onda  
INSERT INTO ... VALUES (GET\_NEXT\_VAL(), 'kita', 'kita') gde ce GET\_NEXT\_VAL biti ovaj gore kod.

## Merging data

**MERGE** statement se koristi da bi mergeovao podatke iz source u target i gde se mogu apply razlicite akcije(INSERT, UPDATE & DELETE) na osnovu "conditional logic".

MERGE je deo SQL standarda i naravno T-sql dodaje nekoliko nonstandard ekstenzija.

Task koji MERGE izvrsava moze takodje da se translira u nekoliko drugih DML statementa bez mergea, ali zasto sebi komplikovati zivot.

Recimo da imas x2 tabele – Customers & CustomersStage i sada trebas da mergeujes contents "CustomersStage"(source) u "Customers"(target). U sustini trebas da dodas customere koji ne postoje i updateujes customers koji postoje.

MERGE je based on "JOIN" semantics. Target table name se specificira u "MERGE", a source table u USING.

!!!Definises merge condition specificiranjem predicatea na "ON" clause. Merge condition definise koji redovi u source tabeli imaju matches u target tabeli. Potom definises "action" koji ces izvrsiti kada je match pronadjen sa WHEN MATCHED THEN i akciju kada match nije pronadjen sa WHEN NOT MATCHED THEN.

Primer gde sa merge se nepostojeci customeri dodavaju i updateuju se postojeci.

```
MERGE INTO dbo.Customers as tgt
USING dbo.CustomersStage as src
ON tgt.custid = src.custid
WHEN MATCHED THEN
UPDATE SET tgt.companyname = src.companyname, tgt.phone = src.phone,
tgt.address=src.address
WHEN NOT MATCHED THEN
INSERT (custid, companyname, phone, address)
VALUES (src.custid, src.companyname, src.phone, src.address);
```

Sve ti je jasno sta se desava.

**MANDATORY je terminateovati MERGE sa “;”!**

Imamo znaci WHEN MATCHED THEN, WHEN NOT MATCHED THEN & T-SQL supports i treći action WHEN NOT MATCHED BY SOURCE THEN koji definise sta se treba desiti kada target row nije matched od strane source row. Znaci ako ga ima u target, a nema u source.

Na primer da zelis da obrises takav row kada nema matching source row.

```
<onaj_query_od_gore>
WHEN NOT MATCHED BY SOURCE THEN
DELETE:
```

Takodje u <onaj\_query\_od\_gore>, update statement se desava pa se desava. Nema nikakav check koji ce doci i reci “updateuj ako se bar jedna vrednost razlikuje”.

Naravno, mozemo to doci i napisati. MERGE statement podrzava dodavanje predicate-a & AND optiona.

```
<target>
<src>
ON <...>
WHEN MATCHED AND
(tgt.companyname <> src.companyname OR tgt.phone <> src.phone OR tgt.address
<> src.address) THEN
UPDATE ....
WHEN NOT MATCHED THEN
...
```

## Modifying data through table expression

Table expressions nisu iskljucivi samo za “SELECT”, mogu ga koristiti i INSERT, UPDATE, DELETE & MERGE jer jelte, to je “reflection of data in underlying tables”.

Znaci da podaci postoje, ali imaju neki svoj izvor.

Modifikovanje podataka kroz table expression ima nekoliko ogranicenja.

- Ako query koji definise table expression joinuje tabele, smes dirati samo jednu stranu joina. Ne mozes obe.
- Ne mozes update kolonu koja je rezultat neke kalkulacije
- INSERT mora specificirati vrednost za sve kolone u underlying table koje ne dobijaju svoje vrednosti “implicitno”. To znaci da ako imas “ROWNUM” kolonu, ona je svoj naziv dobila implicitno npr. Tu ne moras da specificiras.

U sustini use case modifikovanje podataka kroz table expressions je bolji debugging i troubleshooting.

Uzmimo ovaj update statement:

```
UPDATE OD
SET discount += 0.05
FROM dbo.OrderDetails as OD
INNER JOIN dbo.Orders as O ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

Recimo za troubleshooting purposes zelis da vidis koji redovi ce se modifikovati od strane ovog statementa.

Jedan nacin je “revise koda” kroz select statement pa onda runovanje update statementa. Umesto toga, uzece i definisaces table expression na osnovu SELECT statementa sa join query i issue an UPDATE statement nad table expression.

**CTE:**

```
WITH C as
(
    SELECT custid, OD.orderid, productionid, discount, discount + 0.05 as newdiscount
    FROM dbo.OrderDetails as OD INNER JOIN dbo.Orders as O ON OD.orderid = O.orderid
    WHERE custid = 1;
)
UPDATE C
SET discount = newdiscount;
```

### **Derived table:**

```
UPDATE D
SET discount = newdiscount
FROM (SELECT custid, OD.orderid, productionid, discount, discount + 0.05 as newdiscount
      FROM dbo.OrderDetails as OD INNER JOIN dbo.Orders as O ON OD.orderid = O.orderid
      WHERE custid = 1;
) AS D
```

Uz table expression, troubleshooting je mnogo jednostavniji jer uvek mozes samo highlightovati inner select i onda ga izvrsiti bez da se neki podaci "menjaju" jelte.

Nekada je doduse table expression jedina opcija. Za taj use case uzecemo tabelu T1 sa col1 i col2. Stavicemo da su col1 vrednosti 20, 10, 30. col2 je prazan tj NULL.

Recimo da zelis update tabele i postavjanje col2 na rezultat izraza sa ROW\_NUMBER. Problem je sto ROW\_NUMBER nije dozvoljen u SET clause UPDATE statementa.

Window funkcije smeju samo u SELECT ili ORDER BY clause. Dobices error ako pokusas.

```
UPDATE dbo.T1 SET col2 = ROW_NUMBER() OVER (ORDER BY col1);
```

Da bi zaobisao ovaj problem onda uzmes CTE

```
WITH C AS
(
  SELECT col2, ROW_NUMBER() OVER (ORDER BY col1) as rounum FROM dbo.T1
)
UPDATE C
SET col2 = rounum;
```

## **Modifications with TOP and OFFSET-FETCH**

U sustini nisam bas skontao zasto bi zeleo ovo da koristis, ali kao scenario je ako zelis da razdelis posao updatea ili deletea na vise delova/koraka.

Prvi problem na koji ces naici jeste sto TOP i OFFSET-FETCH biraju "prvih x redova" koji se pojavljuju i u insert/update/delete ne mozes uraditi ORDER BY pa ces na:

```
DELETE TOP(50) FROM dbo.Orders -> obrisace prvih 50 redova, bilo kojih 50 redova koji se prvi pojave
```

```
UPDATE TOP(50) FROM dbo.Orders
```

```
SET freight += 10.00;
```

- updatedovace prvih 50 redova koji se pojave

U praksi, nezgodno je updatedovati nesto sto ne znas sta si updateovao i recimo da zelis da imas kontrolu nad tim, onda ces doci i odraditi ordering uz CTE + TOP;

### **WITH C AS**

```
(SELECT TOP(50) * FROM dbo.Orders ORDER BY orderid)
```

### **UPDATE C**

```
SET freight += 10.00;
```

### **WITH C AS**

```
(SELECT TOP(50) * FROM dbo.Orders ORDER BY orderid)
```

### **DELETE FROM C**

Da je OFFSET-FETCH onda bi select bio:

```
SELECT * FROM dbo.Orders  
ORDER BY orderid  
OFFSET 0 ROWS FETCH NEXT 50 ROWS ONLY
```

I sada mozes koristiti "TOP"/"OFFSET-FETCH" u modification queryima tako da su orderovani.

## **OUTPUT Clause**

Statement koji modifikuje ce cesto samo modifikovati. Nekada pozelis da vratis te modifikovane redove za "auditing"/"archiving" ili stagot. Za to sluzi **OUTPUT**

Radi slicno kao SELECT, samo trebas da kazes da li zelis **inserted** keyword za inserted redove ili **deleted** za obrisane redove

OUTPUT vraca result set koji mozes usmeriti u tabelu, onda ces dodati **INTO** clause sa target table name. Takodje mozes output modifikovanih redova vratiti nazad "calleru" i kopiju proslediti u tabelu. Samo napises **x2 OUTPUT** clause-a, jedan sa **INTO** i drugi bez.

## **INSERT with OUTPUT**

Odradicemo sad x1 insert from uz output.

```
INSERT INTO dbo.T1(datacol)  
OUTPUT inserted.keycol, inserted.datacol  
SELECT lastname FROM hr.Employees WHERE Country = 'USA'
```

Zasto bi koristili output. Recimo scenario u kom imamo kolonu u tabeli sa "IDENTITY"-em i sada zelimo da vidimo koji su to sve novonastali dodati ID-evi tj identiteti. OUTPUT ce vratiti te novododane redove, a npr da smo koristili SCOPE\_IDENTITY, onda bi dobili samo poslednji novododani ID jelte. Sad znamo koji id je kom redu dodat.

keycol	datacol
1	Davis
2	Funk
3	Lew
4	Peled
5	Cameron

Da bi output usmerio u neku tabelu ili temp tabelu ili cak variable, mozes OUTPUT INTO

```
DECLARE @NewRows TABLE(keycol INT, datacol NVARCHAR(40));

INSERT INTO dbo.T1(datacol)
OUTPUT inserted.keycol, inserted.datacol
INTO @NewRows(keycol, datacol)
SELECT lastname
FROM hr.employees
WHERE country = 'UK'

SELECT * FROM @NewRows;
```

## DELETE with OUTPUT

U sustini output clause sa delete statement.

```
DELETE FROM dbo.Orders
OUTPUT deleted.orderid, deleted.orderdate, ...
WHERE orderdate < '20220101'
```

Brisne sve porudzbine pre 2022 1. januar

orderid	orderdate	empid	custid
10248	2020-07-04	5	85
10249	2020-07-05	6	79
10250	2020-07-08	4	34
10251	2020-07-08	3	84

## UPDATE with OUTPUT

Koriscenjem OUTPUT-a u UPDATE-u, ti mozes da se pozivas na stanje "row"-a pre modifikacije, a i posle modifikacije.

**deleted** – za pozivanje na state pre changea

**inserted** – za pozivanje na state posle changea

```
UPDATE dbo.OrderDetails  
SET discount += 0.05  
OUTPUT inserted.orderid, inserted.productid  
inserted.discount as new,  
deleted.discount as old  
WHERE productid = 51;
```

orderid	productid	olddiscount	newdiscount
10249	51	0.000	0.050
10250	51	0.150	0.200
10291	51	0.100	0.150
10335	51	0.200	0.250

## MERGE with OUTPUT

Mozes koristiti OUTPUT i uz MERGE, ali znaj da MERGE moze invokeovati vise razlicitih DML actions. Da bi znao koja DML action se izvrsila kako bi se kreirao taj neki output row, koristis funkciju **\$action** unutar OUTPUT clause koji vraca string INSERT/UPDATE/DELETE u zavisnosti od akcije.

```
MERGE INTO dbo.Customers as tgt  
USING dbo.CustomersStage as src  
ON tgt.custoid = src.custid  
WHEN MATCHED THEN  
UPDATE...  
WHEN NOT MATCHED THEN  
INSERT ...  
WHEN NOT MATCHED BY SOURCE THEN  
DELETE  
OUTPUT $action as theaction, inserted.custid, deleted.companyname as  
oldcompanyname, inserted.companyname as newcompanyname;
```

Ako nema nekih vrednosti, onda ce biti samo NULL.

theaction	custid	oldcompanyname	newcompanyname
DELETE	NULL	cust 1	NULL
UPDATE	2	cust 2	AAAAA
UPDATE	3	cust 3	cust 3
DELETE	NULL	cust 4	NULL
UPDATE	5	cust 5	BBBBB
INSERT	6	NULL	cust 6 (new)
INSERT	7	NULL	cust 7 (new)

theaction	custid	oldphone	newphone	oldaddress	newaddress
DELETE	NULL	(111) 111-1111	NULL	address 1	NULL
UPDATE	2	(222) 222-2222	(222) 222-2222	address 2	address 2
UPDATE	3	(333) 333-3333	(333) 333-3333	address 3	address 3
DELETE	NULL	(444) 444-4444	NULL	address 4	NULL
UPDATE	5	(555) 555-5555	CCCCC	address 5	DDDDD
INSERT	6	NULL	(666) 666-6666	NULL	address 6
INSERT	7	NULL	(777) 777-7777	NULL	address 7

(7 rows affected)

## Nested DML

OUTPUT clasue vraca output row za svaki modified row, ali sta ako zelis samo podskup modifikovanih redova u tabelu?

```
INSERT INTO dbo.ProductsAudit(productid, colname, oldval, newval)
SELECT productid, unitprice, oldval, newval
FROM (UPDATE <x> SET unitprice=<x>...OUTPUT <column_names> WHERE supplierid = 1)
WHERE oldval <20 AND newval >= 20
```

U sustini updateuj za suppliera 1 cene, izbaci koje su cene updateovane, filtriraj rezultat i insertuj u dbo.productsaudit. Tim redosledom.

## Exercises do 343.

### Temporal tables

Kada modifikujes podatke u tabeli ti gubis "state" koji je bio u tom "row"-u pre modifikacije.

Sta ako zelis da pratis i previous state podataka za auditing purposes/point in time analysis i slicno? Od MSSQL 2016 postoje **system versioned temporal tables** koje ti to omogucavaju, a mozes u sustini da pravis i svoj customized solution na osnovu triggera(not recommended jer vec imas native resenje)

System versioned temporal table poseduje x2 kolone. Prva koja predstavlja validity period row-a i linked hisotry table koja ima mirrored schema koja sadrzi stariji state modifikovanih redova.

Kad modifikujes podatke, interactujes sa trenutnom tabelom sve najnormalnije. SQL Server automatski radi update za verzije.

Kada treba da vrsis query nad podacima, onda ako zelis trenutni state, queryujes podatke najnormalnije. Ako zelis stariji state, queryujes takodje trenutnu tabelu, ali dadas clause koji indicateuje da zelis "starije podatke".

SQL Standard supports x3 types of temporal tables;

- system versioned; oslanjaju se na system transaction time da definisu validnost(validity) nekog row-a
- application time period tables; oslanjaju se na aplikaciju za definisanje validnosti row-a
- bitemporal; combines the two above(transaction & valid time)

SQL Server 2022 podrzava samo system versioned temporal tables.

## Creating tables

Kad kreiras system versioned temporal table, treba da obezbedis sledece:

- primary key
- x2 columns sa DATETIME2 non nullable koja predstavljaju start & end validity date po UTC time zone
  - o start column treba biti opisana sa GENERATED ALWAYS AS ROW START
  - o end column treba biti opisana sa GENERATED ALWAYS AS ROW END
- oznaciti period columns sa PERIOD FOR SYSTEM\_TIME(<startcol>, <endcol>)
- Opcija tabele SYSTEM\_VERSIONING treba da bude ON
- linked history table(sql server moze da je kreira za tebe) koja sadrzi sva prethodna stanja modifikovanih redova

Od SQL SERVER 2017 mozes definisati **HISTORY\_RETENTION\_PERIOD** koji je subclause SYSTEM\_VERSIONING; vrednosti: INFINITE(default if ommited), DAYS, WEEKS, MONTHS, YEARS. Ako nije infinite, onda sql brise te podatke posle definisanog perioda kroz automatizovani background task.

Ako oznacis period columns sa **HIDDEN** onda prilikom SELECT \* , te kolone se nece vratiti, a kada insertujes data bice ignorisane.

Za kreiranje **Employees** tabele sa linked history table **EmployeesHistory**

```
CREATE TABLE dbo.Employees
(
    empid INT NOT NULL
        CONSTRAINT PK_Employees PRIMARY KEY,
    empname VARCHAR(25) NOT NULL,
    department VARCHAR(50) NOT NULL,
    salary NUMERIC(10, 2) NOT NULL,
    validfrom DATETIME(0)
        GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
    validto DATETIME(0)
        GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,
    PERIOD FOR SYSTEM_TIME(validfrom, validto)
)
WITH ( SYSTEM_VERSIONING = ON
    (HISTORY_TABLE = dbo.EmployeesHistory,
    HISTORY_RETENTION_PERIOD = 5 YEARS));
```

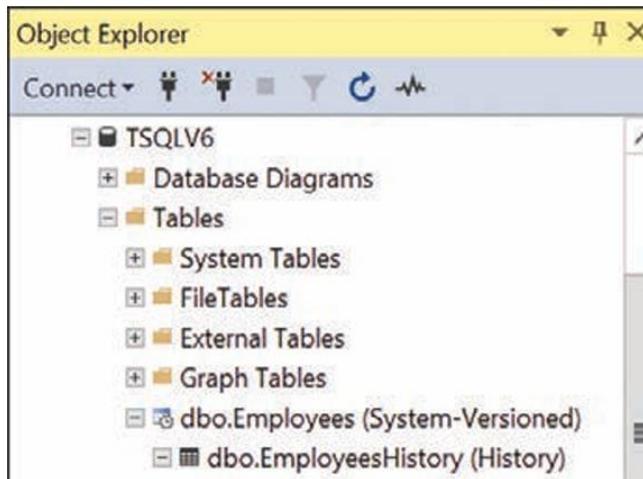
Recimo da EmployeeHistory ne postoji, ako ne specificiras ime onda ce ti MSSQL dodeliti ime tabele -> MSSQL\_TemporalHistoryFor\_<object\_id>, gde je object id, object id trenutne tabele koji mozes dobiti pretrazivanjem object tree u object explorer.

Razlike u automatski generisanoj history tabeli:

- tabela ima mirrored schema trenutne tabele
- nece imati primary key
- clustered index na (<endcol>, <startcol>)
- period columns bez ikakvih posebnih opcija
- bez obelezja period kolona sa PERIOD FOR SYSTEM\_TIME
- history tabela nije marked sa SYSTEM\_VERSIONING

Ako history postoji, mssql samo proveri da li je konzistentnost ispostovana. Ako ne prodje, error na DDLtime. Mozes ozncititi MSSQL-u da ne radi consistency check, ali to se koristi samo kada si 100% siguran da su podaci konzistentni i kada ne mozes da dozvolis vreme neophodno za verifikaciju podataka.

Ako u object explorer exploreujes employees tabelu videces (system-versioned) pored imena.



Takodje mozes non temporal tabele koje vec sadrze podatke pretvoriti u temporalne.  
Recimo stari employees tabela.

```
ALTER TABLE dbo.Employees ADD
validfrom DATETIME2(0) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL
CONSTRAINT DFT_Employees_validfrom DEFAULT('19000101'),
validto DATETIME2(0) GENERATED ALWAYS AS ROW END HIDDEN NOT NULL
CONSTRAINT DFT_Employees_validto DEFAULT('99991231 23:59:59'),
PERIOD FOR SYSTEM_TIME (validfrom, validto);
```

Defaulti postavljaju validity period koji ti mozes da podesavas. Start time je bitno samo da nije u buducnosti, a end day moze da bude maximum supported value u tipu.

Potom alterujes tabelu da enableujes system versioning:

```
ALTER TABLE dbo.Employees
SET (SYSTEM_VERSIONING = ON
(HISTORY_TABLE = dbo.EmployeesHistory, HISTORY_RETENTION_PERIOD = 5 YEARS));
```

NOTE: Ako zelis da vratis hidden polja, onda samo moras da ih dodas u SELECT; znaci SELECT \* ce ih sakriti dok select uz specificne iteme ce ih pokazati.

SQL Server supports menjanje temporal tabele bez da prvo moras da disableujes system versioning. Das mu schema change na current table & sql server applies it to the both current & history tables.

To znači da alter na Employees ce se izvršiti i na employeeshistory.

```
ALTER TABLE Employees  
ADD hiredate DATETIME2(0) NOT NULL  
CONSTRAINT DFT_employees_hiredate DEFAULT('19000101');
```

Ovo ce se dodati i na employeeshistory, ali bez constrainta za default vrednost, ali ako postoje redovi u history table oni ce dobiti hiredate default vrednost prilikom prvih upisa jelte.

Ako dropujemo hiredate; prvo jelte drop constrainta

```
ALTER TABLE employees  
DROP CONSTRAINT DFT_employee_hiredate;
```

```
ALTER TABLE employees  
DROP COLUMN hiredate;
```

SQL ce dropovati kolonu iz obe tabele.

Ako zelis da dropujes system versioned table, prvo moras da disableujes system versioning uz ALTER TABLE komandu i onda mozes rucno dropovati current & history tabele.

## Modyfing data

Modifikovanje privremenih tabela je sличno kao i modifikovanje običnih.

Modifikujes trenutnu tabelu sa insert/update/delete & merge, a u pozadini ce sql odraditi sve za tebe.

Za dalje, doces i insertovaces u employees tabelu iteme:

```
INSERT INTO dbo.Employees(empid, empname, department, salary)  
VALUES(1, 'Sara', 'IT' , 50000.00),  
      (2, 'Don' , 'HR' , 45000.00),  
      (3, 'Judy' , 'Sales' , 55000.00),  
      (4, 'Yael' , 'Marketing' , 55000.00),  
      (5, 'Sven' , 'IT' , 45000.00),  
      (6, 'Paul' , 'Sales' , 40000.00);
```

Napravi upit u obe current & history tabele sa **SELECT**

**current:**

empid	empname	department	salary	validfrom	validto
1	Sara	IT	50000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
2	Don	HR	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	IT	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
6	Paul	Sales	40000.00	2022-02-16 17:08:41	9999-12-31 23:59:59

## history:

<prazna>

To je zato sto se period validnosti smatra "validnim" u trenutku insertovanja i vazi dok je sveta i veka jelte. U momentu kada obrises neki row u current tabeli sa DELETE, onda ce se pored brisanja, uzimati SYSUTCDATETIME() i to ce biti novi "**validto**" u history tabeli.

Dakle, brisanje itema u current se pretvara u insert u history. Da si uradio UPDATE na current i povecao plate za 10%, odradio bi se insert u history za sve redove koji su zahvaceni.

!Ukoliko ti nije bitno da drzis state modifikovanih redova(older state of modified rows) za time travel, ali zelis da znas kada se neki row insertovao ili updateovao onda ces temporal tabele da kreiras sa SYSTEM\_VERSIONING OFF opcijom i bez history tabele. I dalje ces morati imati period start & end kolone. Potom kako insertujes i updateujes redove u tabelu, sql server zapamti taj moment i upise ga.!

**Modification times** koje ce sql server da cuva jeste pocetak transakcije. Ako ti transakcija dugo traje, to znaci da ce se cuvati T1(transaction start time) svuda.

Recimo ako si okinuo transakciju(UPDATE recimo) u 17:28:10, to ce za novu vrednost u current tabeli biti **validfrom**, a u history tabeli **validto** za staru vrednsot.

To moze da bude i problematicno u slucaju da ako updateujes row vise puta unutar iste transakcije i zavrsices sa "in-between" verzijama koje imaju zero length validity periods u kojima **validfrom** & **validto** ce biti identicno.

Takvi intervali se nazivaju *degenerate intervals*.

Kada pravis upit nad temporal tabelama sa FOR SYSTEM\_TIME clause, automatski ce se odbaciti degenerate interval rows.

Scenario sledeci: imas transakciju u transakciji gde obe updateuju isti row.

TABLE 9-1 Noteworthy modification sequence

point in time	session 1	session 2
T1	Begin tran A	
T2		Begin tran B
T3		Update some row R in a system-versioned table (recorded modification time is T2)
T4		Commit tran B
T5	Update the same row R (recorded modification time is T1, which is earlier than the last recorded modification time T2!)	
T6	Commit tran A	

U vremenu T5 ce da puca sa errorom “data modification failed on system-versioned table jer transaction time je u proslosti tj pre nego start time validnosti”.

SQL Server ce drzati transaction A otvorenu tako da bi valjalo imati error handling prisutan kada naidjes na ovakve scenarie.

## Querying data

Slanje upita temporal tabelama; ako zelis state current podataka onda queryujes trenutnu tabelu. Ako zelis past state, ti i dalje queryujes current tabelu, ali dodas clause **FOR SYSTEM\_TIME** i podupit koji nalaze na validnost perioda koji zelis.

<sad jedan refresh svih tabela>

Ukoliko queryujes viewove sa FOR SYSTEM\_TIME, clause ce se dalje propagirati na underlying objekte.

**SELECT <> FROM <table/view> FOR SYSTEM\_TIME <subclause> AS <alias>**

**for system\_time as of**

Najcesce ces korisiti “AS OF” subclause koji: FOR SYSTEM\_TIME AS OF <datetime2>

Input moze biti constant, variable ili parametar

Rezultat koji ce ti dati select jesu redovi koji ispunjavaju kriterijum u kom je taj datetime iz as of clause posle “validfrom”, ali pre “validto”

**validfrom <= datetime AND validto > datetime**

```
SELECT * FROM dbo.Employees  
FOR SYSTEM_TIME AS OF '2022-02-16 17:00:00'
```

<dobijas prazan result set jer recimo da si sve unosio u 17:08:00>

```
SELECT * FROM dbo.Employees  
FOR SYSTEM_TIME AS OF '2022-02-16 17:10:00'
```

empid	empname	department	salary
2	Don	HR	45000.00
4	Yael	Marketing	55000.00
6	Paul	Sales	40000.00
1	Sara	IT	50000.00
5	Sven	IT	45000.00
3	Judy	Sales	55000.00

Mozes queryovati visestruke instance iste tabele gde ces poreediti razlicite stateove podataka u razlicitim vremenskim periodima. Npr query koji vraca procenat povecanja plate zaposlenih koji su imali salary increase izmedju dva razlicita point in time.

```
SELECT T2.empid, T2.empname,
CAST( (T2.salary/T1.salary - 1.0) * 100 as NUMERIC(10,2) ) AS pcnt
FROM dbo.Employees FOR SYSTEM_TIME AS OF '2022-02-16 17:10:00' as T1
INNER JOIN dbo.Employees FOR SYSTEM_TIME AS OF '2022-02-16 17:25:00' as T2
ON T1.empid = T2.empid AND T2.salary > T1.salary;
```

empid	empname	pct
1	Sara	5.00
5	Sven	5.00

AS OF <datetime> znaci uzima sve recorde od pocetka vremena pa do te tacke u vremenu.

from @start to @end

Subclause **FROM @start TO @end** vraca redove koji ispunjavaju:

validfrom < @end AND validto > @start

Drugim recima, vracaju se redovi koji su bili validni u periodu izmedju @start & @end

```
SELECT *, validfrom, validto
FROM dbo.Employees
FOR SYSTEM_TIME FROM '2022-02-16 17:15:26' TO '2022-02-16 17:20:02'
```

empid	empname	department	salary	validfrom	validto
2	Don	HR	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10

from @start to @end nije inkluzivan. Ukoliko zelis da include @start i @end onda je bolje koristiti sledece:

## between @start and @end

SELECT ... FROM ...

FOR SYSTEM\_TIME BETWEEN '2022-02-16 17:15:26' AND '2022-02-16 17:20:02';

empid	empname	department	salary	validfrom	validto
1	Sara	IT	52500.00	2022-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10
5	Sven	IT	47250.00	2022-02-16 17:20:02	2022-02-16 17:28:10

contained in(@start, @end)

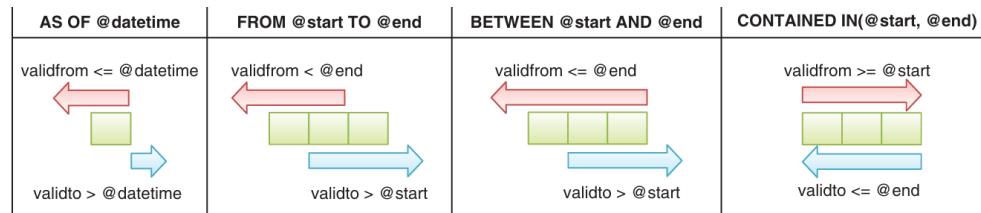
Vraca redove gde validfrom >= @start AND validto <= @end

Drugim recima, redovi koji su poceli da vrede posle @start, a prestali da vrede pre @end.  
Inclusive.

```
SELECT empid, empname, department, salary, validfrom, validto
FROM dbo.Employees
FOR SYSTEM_TIME CONTAINED IN('2022-02-16 17:00:00', '2022-02-16 18:00:00');
```

This query generates the following output:

empid	empname	department	salary	validfrom	validto
6	Paul	Sales	40000.00	2022-02-16 17:08:41	2022-02-16 17:15:26
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10
5	Sven	IT	47250.00	2022-02-16 17:20:02	2022-02-16 17:28:10



### Legend:

-  @datetime Input date and time value
-  @start @end Input start and end date and time values
-  validfrom System period start column
-  validto System period end column

all

Hah dobijes rezultate iz svih tabela

**SELECT ... FROM ... FOR SYSTEM\_TIME ALL**

Exercises do 366

## Transactions and concurrency

Objasnjava transakcije, konkurentnost i sta se desava kada vise korisnika pokusa da pristupi istim podacima.

Posto je ovo fundamentals, uzece samo scenarie u kom se obradjuju podaci koji su “u disku”. Postoji i memory-optimized tabela koja radi na memory optimizes InMemory OLTP koji je van scopea ove knjige.

## Transactions

Transakcija je unit of work koja moze ukljuciti visetruke aktivnosti za upite i modifikacije podataka i takodje moze da menja “data definition”.

BEGIN TRAN

ROLLBACK TRAN/TRANSACTION

COMMIT TRANSACTION

Ako ne označis eksplisitno svoje upite unutar transakcije, sql server ce tretirati svaki upit/statement kao transakciju. To se naziva “auto-commit” mode koji mozes menjati naravno sa iskljucivanjem/ukljucivanjem implicit transakcija nad sesijom; opcija je IMPLICIT\_TRANSACTIONS

Kada su implicitne transakcije ukljucene, onda ne moras zapocinjati transakcije sa BEGIN TRANSACTION, ali moras ih zavrsavati sa COMMIT TRAN/ROLLBACK TRAN.

## ACID

Transakcije poseduju x4 svojstva – atomicnost, konzistentnost, izolacija i izdržljivost.

**Atomicity** – transakcija je atomicni unit of work. Ili ce se sve promene izvrsiti ili ni jedna.

Kod atomicnosti, neke stvari se ne smatraju “tolikim problemom” kao npr primary-key violations & lock-expiration timeouts pa se nece dogoditi automatic rollback transakcije.

Ako zelis da sve greske izazovu rollback onda enableujes session **XACT\_ABORT**

**Consistency** – odnosi se na stanje podataka kojima ti rdbms daje pristup kao deo konkurentne transakcije. U sustini ona prica sa isolation levels. Consistency se takodje

odnosi da baza mora pratiti pravila constraintova. Transactions transition the db from one consistent state to another.

**Isolation** – Isolation ensures da transakcije pristupaju samo consistent podacima. Ti kontrolises sta konzistentnost znaci u tvojim transakcijama kroz izolacije(isolation levels). Kod disk based tabela postoje x2 modela izolacija. Jedna koja se bazira na lockovanju, a druga kombinaciji lockovanja i row versioninga(bice referred as row versioning samo).

**Durability** – Znaci da jednom kada se transakcija commitovala, promene koje su izvrse su “durable” to jest zadrzane u bazi. Commit je acknowledged by getting control back to the application and running the next line of code.

Mehanizam koji garantuje durability zavisi od toga koji je recovery architecture postavljen u bazi. (page 369)

- ➔ Ako pises transakciju i zelis da reagujes na “greske”, onda koristis TRY/CATCH blokove.

## Locks & Blocking

Po defaultu SQL Server box product koristi pure locking model kako bi nametnuo izolaciju.

Azure SQL Database koristi row-versioning model po defaultu. Ako si na SQL DB iz Azure onda moras iskljuciti READ\_COMMITTED\_SNAPSHOT.

**ALTER DATABASE TSQLV6 SET READ\_COMMITTED\_SNAPSHOT OFF;**

Ako si na vec toj bazi onda umesto TSQLV6 stavis **CURRENT**. Svedno je svakako.

## Locks

Locks su “control resources” koji se dobijaju od transakcije kako bi cuvali “data resources”, vrse prevenciju conflicta ili inkompatibilnog pristupa drugih transakcija.

## Lock modes & compatibility

Postoje dva glavna lock moda: **exclusive & shared**

Kad pokusavas modifikovati podatke, tvoja transakcija zahteva exclusive lock nad data resursom bez obzira na tip nivoa izolacije. Ako se lock dobije, on se zadrzava do isteka/zavrsetka te transakcije(bila ona single ili multistatement).

Exclusive se naziva jer ne mozes dobiti exclusive lock nad resursom ako druga transakcija trenutno radi nesto. Ne mozes nikakav lock dobiti generalno ako imas exclusive lock(default ponasanje i ne moze se menjati).

E sada, ono sto moze da se desi jeste da druga transakcija dodje i cita podatke nad kojima je postavljen lock. To je nesto o cemu govori **isolation level**.

U SQL Server default isolation level je READ COMMITTED

U Azure sql db je READ COMMITTED SNAPSHOT.

READ COMMITTED znaci da kad pokusas da citas podatke, dobijas shared lock nad objekt resursom i lock se releasuje cim je read gotov. Naziva se "shared" jer vise transakcija moze da ga deli istovremeno.

READ COMMITTED SNAPSHOT je kombinacija lockovanja i row versioninga. Na ovaj nacin readers do not require shared locks pa s toga nikada ne cekaju. Oslanjaju se na row versioning tehnologiju da dostavi "ocekivanu izolaciju".

U praksi to znaci da ako pod READ COMMITTED neka transakcija modifikuje row, sve dok se transakcija ne zavrsi, druga transakcija ne moze citati iste redove. Taj pristup konkurentnosti se naziva "*pessimistic concurrency*".

Pod READ COMMITTED SNAPSHOT, ako transaction modifikuje redove, druga transakcija koja pokusava da cita podatke ce dobiti poslednje commitovani state koji je dostupan. Taj pristup konkurentnosti se naziva "*optimistic concurrency*".

Ovakva lock interakcija izmedju transakcija se naziva *lock compatibility*. Tabela ispod pokazuje lock compatibility exclusive & shared lockova. Redovi su requested lock modes, a kolone granted lock modes.

**TABLE 10-1** Lock compatibility of exclusive and shared locks

Requested mode	Granted Exclusive (X)	Granted Shared (S)
Exclusive	No	No
Shared	No	Yes

"No" u preseku znaci da lockovi nisu kompatibilni i da je requested mode denied; the requester must wait. Yes znaci da su kompatibilni.

Summary lock interakcija izmedju transakcija:

Podaci koji su modifikovani od jedne transakcije ne mogu biti modifikovani, a ni pročitani od strane druge transakcije dok se prva transakcija ne zavrsi, a dok se podaci citaju od strane jedne transakcije, ne mogu biti modifikovani od strane druge.

## Lockable resource types

U sustini razliciti tipovi resursa se mogu lockovati. To su rows(RID ako je heap, key ako je index), pages, objects(npr tabele), databases i drugo.

Rows su unutar stranice(pages). Pages su fizicki blokovi podataka koji sadrze table ili index data.

Familiarize yrslf sa extens, allocation units & heaps(b-trees)

Da bi dobio lock na neki resource type, transakcija mora prvo dobiti "intent locks" istog moda ili viseg nivoa slozenosti. Znaci da ako transakcija zeli exclusive lock na row, prvo mora dobiti intent exclusive locks nad table & page gde se row nalazi. Isto vazi i za shared lock.

Svrha "intent lock-a" je da efikasno detektujes inkompatibilne lock requestove na visim nivoima slozenosti i da onesposobis da doiju grant. Npr ako x1 transaction drzi lock na row-u, a druga pita za inkompatibilan lock mode na ceo page/table gde se taj row nalazi, jako je lako da sql server identifikuje conflict jer intent lock koji je prva transakcija dobila. Intent lock se ne sukobljava za sequestovima za lockove na nizim nivoima slozenosti sto znaci da ako imas intent lock nad page, to nece spreciti intent lock nad row iz druge transakcije ako se taj row nalazi u toj page.

TABLE 10-2 Lock compatibility including intent locks

Requested mode	Granted Exclusive (X)	Granted Shared (S)	Granted Intent Exclusive (IX)	Granted Intent Shared (IS)
Exclusive	No	No	No	No
Shared	No	Yes	No	Yes
Intent Exclusive	No	No	Yes	Yes
Intent Shared	No	Yes	Yes	Yes

SQL Server odlucuje fizicki o tome koji resource type ce lockovati. Za idealnu konkurentnost naravno najbolje je lockovati samo ono sto treba da se lockuje tj only the affecting rows.

Lockovi zahtevaju memorije i internal management overhead tako da razmisli o tome kad odlucis da lockujes i sta da lockujes.

Kad SQL Server estimira da ce transakcija interactovati sa malim brojem redova, onda se koristi uglavnom row lock dok kada ce transakcija imati uticaj na veci broj redova onda page lock.

- prica o nekim fine grained locks na stranici 373, nisam skontao

Mozes podesiti LOCK\_ESCALATION koriscenjem ALTER TABLE statementa da kontrolises lock escalation behavior ili da ga iskljucis.

## Troubleshooting blocking

Kad jedna transakcija zatrazi incompatible lock nad resursom koji je vec lockovan od druge transakcije, onda ceka da lock istekne. Mozes definisati lock expiration u sesiji ako zelis da kontrolises vreme cekanja pre time-outa locka.

Blocking je normalno OSIM ukoliko ne unosi previse laga u sistem. Npr ako imas long running transakcije mozes probati da ih razbijes u vise i izmestis iz jednog “unit of work”. Bug u aplikaciji moze da zadrzi transakciju otvorenu duze nego sto bi trebalo

Npr sad da si na READ COMMITTED i da u jednoj sesiji otvoris transakciju bez da zatvoris  
BEGIN TRAN;

```
UPDATE Production.Products
    SET unitprice += 1.00
    WHERE productid = 2;
```

I u drugoj sesiji posaljes upit na taj row koji je i dalje u transakciji

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Dobijes prc. U sesiji 2 treba ti shared lock kako bi procitao, ali jer je row exclusively locked, a shared lock je inkompatibilan sa exclusive lock, sad ti je sesija 2 blokirana i mora da ceka.

Kako bi troubleshoot-ovao sad ces raditi query nad **dynamic management objects** koji ukljucuju podatke o raznim delovima tvog sistema poput viewova, funkcija i slicno.  
Izvrsavace se iz sesije 3.

**Getting lock info**, ukljucujuci locks koji su trenutno “u akciji” i lockove na koje druge sesije cekaju; queryuj (**DMV**) sys.dm\_tran\_locks

```
SELECT ... FROM sys.dm_tran_locks;
```

```

SELECT -- use * to explore other available attributes
    request_session_id      AS sid,
    resource_type            AS restype,
    resource_database_id     AS dbid,
    DB_NAME(resource_database_id) AS dbname,
    resource_description     AS res,
    resource_associated_entity_id AS resid,
    request_mode              AS mode,
    request_status             AS status
FROM sys.dm_tran_locks;

```

CHAPTER 10 Transactions and concurrency

When I run this code in my system (with no other query window open), I get the following

sid	restype	dbid	dbname	res	resid	mode	status
53	DATABASE	8	TSQLV6		0	S	GRANT
52	DATABASE	8	TSQLV6		0	S	GRANT
51	DATABASE	8	TSQLV6		0	S	GRANT
54	DATABASE	8	TSQLV6		0	S	GRANT
53	PAGE	8	TSQLV6	1:127	72057594038845440	IS	GRANT
52	PAGE	8	TSQLV6	1:127	72057594038845440	IX	GRANT
53	OBJECT	8	TSQLV6		133575514	IS	GRANT
52	OBJECT	8	TSQLV6		133575514	IX	GRANT
52	KEY	8	TSQLV6	(020068e8b274)	72057594038845440	X	GRANT
53	KEY	8	TSQLV6	(020068e8b274)	72057594038845440	S	WAIT

sid – session id; svaka sesija ima session id. ID tvoje sesije mozes dobiti queryovanjem **@@SPID funkcije** ili u zagradi pored connection name-a u footeru ssms-a.

U gore otpisu mozes da vidis da session 53 ceka na shared lock. Sesija 52 drzi exclusive lock nad istim row-om. Mozes posmatranjem skontati da oba sessiona lockuju row unutar istog res & resid value. Mozes skontati koje su tabele ukljucene tako sto ces se pomerati gore u lock hierarchy za sesije 52/53 i ledati intent locks nad objektom/tabelom gde se row nalazi.

Mozes koristiti OBJECT\_NAME function da transliras object id iz resid atributa object locka da bi skontao da je u pitanju *Production.Products* tabela.

sys.dm\_tran\_locks view ti daje info o ID-evima sesija koje su ukljucene u blocking chain. **blocking chain** je lanac od 2 ili vise sesije koje su ukljucene u “blocking situation”.

Da dobijes info o konekcijama koje su povezane sa session id-evima, pozivas se na *sys.dm\_exec\_connections* i filtriras samo session ID's koji su ukljuceni.

```

SELECT -- use * to explore
    session_id AS sid,
    connect_time,
    last_read,
    last_write,
    most_recent_sql_handle
FROM sys.dm_exec_connections
WHERE session_id IN(52, 53);

```

Znaci session ID's koji su ukljuceni u blocking chainu u ovom slucaju su 52 i 53.

```
sid connect_time           last_read
---- -----
52   2022-06-25 15:20:03.360 2022-06-25 15:20:15.750
53   2022-06-25 15:20:07.300 2022-06-25 15:20:20.950

sid last_write             most_recent_sql_handle
---- -----
52   2022-06-25 15:20:15.817 0x01000800DE2DB71FB0936F05000000000000000000000000000000
53   2022-06-25 15:20:07.327 0x0200000063FC7D052E09844778CDD615CFE7A2D1FB411802
```

Ovo dobijes kao rezultat od info.

connection\_time je vreme kada su se konektovale(sesije)  
last read & last write je poslednje vreme upisa i citanja  
potom imamo binarnu vrednost koja drzi "most recent" sql batch run by the connection. Taj handle das kao input funkciji sys.dm\_exec\_sql\_text & funkcija vraca batch koda represented by the handle.

U prevodu, dobijes kod koji se izvrsio da bi se jelte ostvarilo to "stanje"

```
SELECT session_id, text
FROM sys.dm_exec_connections
CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(52, 53);

session_id  text
-----
52          BEGIN TRAN;

                      UPDATE Production.Products
                      SET unitprice += 1.00
                      WHERE productid = 2;

53          (@1 tinyint)
          SELECT [productid],[unitprice]
          FROM [Production].[Products]
          WHERE [productid]=@1
```

Da dobijes isti rezultat od mssql 2016 mozes koristiti:

sys.dm\_exec\_input\_buffer(session\_id, NULL) i onda bi sa tim odradio cross apply za isti rezultat.

Unutar **sys.dm\_exec\_sessions** mozes svasta o sesiji dobiti poput login time, host name, program name, login name, nt\_user\_name, session id...

Unutar dmv-a **sys.dm\_exec\_requests** pronacices row za sve aktivne requestove ukljucujuci i blocked requests. Ako je blocking\_session\_id veci od 0 onda je to blocking request. U tom DMV mnogo lakse mozes dobiti info o blokadama.

Odradi select \*, ali u sustini wait type, db id, command, sql handle, session id, blocking session id, wait resource...

sid	login_time	host_name
52	2022-06-25 15:20:03.407	K2
53	2022-06-25 15:20:07.303	K2

sid	program_name	login_name
52	Microsoft SQL Server Management Studio - Query	K2\Gandalf
53	Microsoft SQL Server Management Studio - Query	K2\Gandalf

sid	nt_user_name	last_request_start_time	last_request_end_time
52	Gandalf	2022-06-25 15:20:15.703	2022-06-25 15:20:15.750
53	Gandalf	2022-06-25 15:20:20.693	2022-06-25 15:20:07.320

Unutar **sys.dm\_os\_waiting\_tasks** imas takodje blocking session id.

## Unblocking a session

**NAPOKON – da bi odblokirao sesiju:**

**KILL <session\_id>**

Napomena: po defaultu session nema lock timeout set.

SET LOCK\_TIMEOUT 5000(vrednost u milisekundama, ovo je 5s)

0 – immediate timeout

(-1) – no timeout(default)

SET LOCK\_TIMEOUT 5000

<query 2 koji trazi productid = 2 nad kojim je lock>

➔ Posle 5 sec ce mssql izbaciti:

Msg 1222, Level 16, State 51, Line 3  
Lock request time out period exceeded.

**SET LOCK\_TIMEOUT -1;** vratis ga nazad jelte

KILL 52 -> rollback transakcije u connection 1.

## Isolation levels

Nivoi izolacije odlucuju o nivou konzistentnosti koji dobijas kada interactujes sa podacima.

Po defaultu kad se desava read, koristi se shared lock nad resursom, a kada je write onda exclusive lock. Ono sto mozes kontrolisati jeste kako se readeri ponasaju, ali ne kako i writeri u smislu lockova koji dobiju, ali mozes implicitno uticati na writere.

SQL Server nudi x4 opcije izolacije:

READ UNCOMMITTED, READ COMMITED(default box), REPEATABLE READ & SERIALIZABLE.

Takodje nudi x2 nivoa izolacije koji se baziraju na kombinaciji lockovanja i row versioninga:  
SNAPSHOT & READ COMMITED SNAPSHOT(default azure sql)

Mozes podesiti isolation level cele sesije sa:

**SET TRANSACTION ISOLATION LEVEL <isolation name>**

Mozes koristiti table hint da podesis isolation level query-a

**SELECT ... FROM <table> WITH (<isolationname>)**

READ COMMITED SNAPSHOT ne moze eksplisitno da se postavi kao isolation level nad sesijom/queryem. Da bi to omogucio onda moras podesiti poseban db flag.

Sto je veci lock/isolation level, to je manja konkurentnost, ali veca konzistentnost.

### READ UNCOMMITTED

READ UNCOMMITTED je najnizi nivo izolacije gde reader ne pita za shared lock pa samim tim i ne moze biti u conflictu sa writerom. To znaci da reader moze da procita uncommitted changes(dirty reads).

### READ COMMITED

Ako zelis da preventiras readere od citanja uncommitted changes, onda koristis ovaj jaci isolation level.

To radi tako sto od readera ocekuje da pribavi shared lock. Duration of the lock kod read committed, reader drzi shared lock samo dok ne zavrsi sa resursom. Ne drzi ga dok se zavrsi transakcija. Znaci da svaki read unutar iste transakcije ce u sustini imati lock za sebe.

To znaci da druga transakcija(ako se zadesi) moze da promeni vrednost resursa izmedju x2 read-a. Taj fenomen naziva se *nonrepeatable reads* ili *inconsistent analysis*.

## REPEATABLE READ

Ako zelis da zagarantujes da niko nece promeniti vrednosti izmedju readova unutar iste transakcije, mozes podesiti isolation level na REPEATABLE READ. Ovde reader drzi shared lock sve do kraja transakcije sto znaci da cim reader acquireuje shared lock(koji traje do kraja transakcije), niko ne moze da dobije exclusive lock nad objektom dok se transakcija ne zavrsi.

```
SET ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN TRAN;
```

```
...
```

## SERIALIZABLE

U repeatable read se lockuju samo resursi(npr rows) koje je query pronasao prvi put kad se izvrsio, ali ne i redove koji su se pojavili "kao novi redovi". Ti novi redovi su "phantoms" i takve readove nazivamo phantom reads, da izmedju readova(rowova) druga transakcija insertuje nove redove koji ispunjavaju kriterijume tvog readera.

Serializable je nivo izolacije iznad repeatable reada.

Isto kao repeatable read drzi shared lock sve do kraja transakcije, ali ubacuje facet – logicki, da se lockuje ceo range of keys koji ispunjavaju query filter. To znaci da ce reader lockovati trenutne, a i buduce rows.

- odradi select sa set isolation level serializable; dobijes 12 redova npr, nemoj zatvoriti transakciju(BEGIN TRAN)
- Odradi iz druge sesije insert. Attampt je blocked jer insert ce biti nizeg nivoa izolacije od serializable
- odradi opet select, dobices opet 12 redova. Zatvorи transakciju (COMMIT TRAN)
- Sada sesija druga moze dobiti exclusive lock i ako ponovis ove selecte, dobices 13 redova.

## Isolation levels based on row versioning

Uz row versioning tech, sql server storeuje prethodne verzije committed rowsa u version store. Taj version store je u *tempdb* osim ukoliko nije ukljucen ADR(Accelerated database recovery) i onda ce se version store storeovati u database in question.

SNAPSHOT isolation level je slican SERIALIZABLE isolation levelu u smislu tipova consistency problema koji se mogu ili ne mogu dogoditi.

READ COMMITTED SNAPSHOT isolation level je slican READ COMMITTED isolation levelu. Readeri koji koriste isolation levels na osnovu row versioninga ne uzimaju locks pa ne moraju da cekaju kad se desi neki exclusive lock.

Sa versioning sql server daje starije verzije row-a ako trenutna verzija nije ona koju treba da vide.

Ako enableujes row-versioning based isolation levels, DELETE & UPDATE statements moraju kopirati verziju row-a pre nego izmene version store. INSERT ne mora da upisuje nista u version store jer jelte ne postoje prethodne verzije row-a.

To znaci da ce update-i i delete-i biti sporiji. Performance readera ce se (nekada)drasticno poboljsati.

### SNAPSHOT isolation level

SNAPSHOT kad reader readuje podatke, garantuje da ce dobiti poslednji committed version row-a koji je dostupan pre nego je transakcija zapocela. To garantuje repeatable reads nad committed reads.

1. da bi ovo enable

```
ALTER DATABASE TSQLV6 SET ALLOW_SNAPSHOT_ISOLATION ON;
```

To znaci da kada zapocnes transakciju sa READ COMMITTED gde ces staviti lock nad resursom, opet ce se poslednja verzija zapisati u version store.

Ako sesija 1 ima otvorenu transakciju prema row gde nek je price 19 pre transakcije, a postalja se na 22 u transakciji. Potom imamo sesiju 2 koja cita iz istog row-a, ta sesija ce dobiti poslednju committed vrednost i ne treba joj shared lock i procitace 19.

Iako je sesija 1 u procesu "menjanja" te vrednosti, druge sesije mogu da citaju poslednju committed vrednost.

sesija 1:

```
BEGIN TRAN;  
UPDATE <ta neka tabela> SET price = 22 WHERE id = 5;
```

sesija 2:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

```
BEGIN TRAN
```

```
SELECT price FROM <ta neka tabela>
```

## Conflict detection

SNAPSHOT isolation level preventira update conflicts i izbaci error umesto da generise deadlock kao repeatable read & serializable.

Update conflict se desi inspectovanje version store-a. Moze skontati ako je neka transakcija modifikovala podatke izmedju read-a i writea koji su se desili u twojоj transakciji.

Kako se to moze dogoditi:

### **session 1:**

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRAN;  
SELECT * FROM products WHERE id = 2;
```

potom query 2 u session 1  
UPDATE products SET price = 20 WHERE id = 2;  
COMMIT TRAN;

I ovo nece nikakav exception baciti.

Da si runnovao query 2 u session 2 pa se onda vratio i pokusao da ga updateujes u session 1(tj isti update query u session 1) dobio bi:

```
Msg 3960, Level 16, State 2, Line 1  
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation  
to access table 'Production.Products' directly or indirectly in database 'TSQLV6' to update,  
delete, or insert the row that has been modified or deleted by another transaction. Retry the  
transaction or change the isolation level for the update/delete statement.
```

Mozes uz error handling retry ceo transaction.

Ovo je u sustini samo error koji se dogodi pa da znas :)

## READ COMMITTED SNAPSHOT

bazira se na row versioning. Za razliku od snapshot-a gde se reader provideuje sa transaction level consistent view, ovde se provideuje reader sa statement level consistent view. U sustini za svaki select ti ces uzimati poslednju committed verziju pa moze da se desi npr neki update izmedju dva selekta.

READ COMMITTED SNAPSHOT ne detectuje update conflicts. Readeri ne uzimaju shared lock i ne cekaju na exclusively locked resource da bude unlocked. Samo citaju iz version storea.

Ako imas read committed snapshot selektovan i ako zelis da se zauzme shared lock nad resursom, moras dodati table hint READCOMMITTEDLOCK;

`SELECT * FROM ... WITH (READCOMMITTEDLOCK)`

Znaci da bi koristio ovaj isolation level moras

`alter database <naziv> SET READ_COMMITTED_SNAPSHOT ON;`

Ono sto ce se sada desiti jeste da on postaje default i umesto da ti svaki query ide na READ COMMITED, on ce se prebaciti na READ COMMITED SNAPSHOT osim ukoliko ne postavis eksplicitno isolation level sesije.

U sustini imas session 1, ides begin tran i updateujes price na 20.

Potom u istoj sesiji odradis select i dobices value sa 20 jer si u istoj transakciji koja jos nije zavrsena

Ukoliko uzmes sesiju 2 i odradi select, ti ces dobiti startu vrednost price-a od 19.

Tek kad commitujes prvu transakciju, tad ces moci citati version table u sesiji 2.

page 393; procitaj ponovo. Izgoreo si.

## DEADLOCKS

Deadlock je situacija u kojoj dva ili vise sesija blokiraju jedna drugu. SQL reaguje na deadlockove tako sto u trenutku prepoznavanja deadlocka, terminateuje jednu od transakcija da ne bi vecno ostali u deadlocku. Terminateuje onu koja je uradila najmanje posla na osnovu aktivnosti po transaction logu(rollback te transakcije ce biti najjeftiniji).

Podesavanjem DEADLOCK\_PRIORITY vrednostima -10 do 10(21 vrednost) mozes podesiti kako se bira "zrtva" gasenja. Uglavnom sesija sa najnizim deadlock priority se bira kao "zrtva" bez obzira na kolicinu posla. Ukoliko neke sesije dele isti deadlock priority, sql server bira random sesiju.

page 395 gde daje code example, nista spec.

Msg 1205, Level 13, State 51, Line 1

Transaction (Process ID 52) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Error za deadlock.

Dobar index dizajn moze skratiti dosta overheada koji izaziva deadlockove. Deadlockovi se desavaju cesto zbog logickog konflikta jer npr sql mora lockovati celu tabelu i skenirati rows ako nemas dobro postavljen index.

READ COMMITED SNAPSHOT je gde readerima ne treba shared lock i deadlocks koji nastaju usred shared locks nestaju.

## SQL Graph

Predstavljeno u sql 2017, u sustini predstavlja drugaciji pristup modelovanju i slanju upita podacima.

Uglavnom uključuje **nodes & edges**. Node je tacka u graphu, a edge je veza koja stvara "par" nodeova.

Querying ima posebnu sintaksu koja mora koristiti i MATCH clause da govori o tome kako se nodeovi povezuju.

MSFT implementation of cypher language. Skipping this as there are better solutions.  
Spend time into other resources or come back to this only if necessary.

Oko stranice 400 do 490, kasnije baci pogled na to.

## Programmable objects

Ovde se prica o variables, batches, flow elements, cursors, dynamic SQL, routines(user defined fncts, stored procedures & triggers) and error handling.

➔ istrazi na 491. stranici o JSON & XML

### Variables

Mozes koristiti promenljive da privremeno cuvas data values za kasnije koriscenje.

DECLARE za declareovanje 1 ili vise variables.

SET za postavljanje vrednosti x1 vrednosti.

**DECLARE @i AS INT;**

**SET @i = 10;**

Alternativno mozes u istoj liniji deklarisati i dodeliti vrednost:

**DECLARE @i AS INT = 10;**

Mozes i scalar subquery umesto konstante(10).

Potom ga ovako prikazes npr:

**SELECT @i AS counter;**

**DECLARE @i AS INT, @lastname AS NVARCHAR(60);**

I sada ako nesto od toga selectujes bez da si ga **SET**(moze samo x1 variable at a time), onda dobijes **NULL**

## Batches

Batch je x1 ili vise T-SQL statementa koje se salju od strane client aplikacija ka SQL serveru za izvrsavanje kao “single unit”. Batch mora da prodje “parsing”(syntax checking), resolution/binding(provera da li referencirani objekti postoje) i optimization as a unit.

Batch nije transaction. Transaction je atomic unit of work dok batch moze da ima vise transakcija. Primer batch-a jeste kad npr ado.net salje neki “sql” ka bazi i baza znaci mora prvo da ga primi, prevede, proveri i optimizuje pre nego ga izvrsi. Stored procedure se izvrsavaju npr natevely. Ovo ne.

**GO** komanda govori da je ovo sada “kraj batcha” i to je client tool command, a ne t-sql server command. Najbolje je GO ne zavrsavati sa “;” jer to nije deo t-sql sintakse.

### Batch as a unit of parsing

Ako je prevodjenje uspesno, sql server potom pokusava da izvrsi taj batch. U slucaju syntax errora, ceo batch se odbacuje.

To znaci da ako posaljes x3 batcha i ovaj drugi ima syntax error, onda ce se ovi ispravni izvrsiti dok ovaj sa greskom nece

```
Print 'First batch'; SELECT * FROM... –valid batch  
GO
```

```
Print 'Second batch'; Select * FROM...; SELEC * FOM... –invalid batch  
GO
```

```
Print 'Third batch'; SELECT * FROM... –valid batch  
GO  
First batch  
Msg 102, Level 15, State 1, Line 91  
Incorrect syntax near 'Sales'.  
Third batch  
empid  
-----  
2  
7
```

## Batches and variables

Variable je local batchu u kom se definise.

```
DECLARE @a AS NVARCHAR(12) = 'moja kita';(10)
PRINT @a;
GO

PRINT @a; --Fails
GO

10
Msg 137, Level 15, State 2, Line 106
Must declare the scalar variable "@i".
```

### Statements that cannot be combined in the same batch

```
CREATE DEFAULT
CREATE FUNCTION
CREATE PROCEDURE
CREATE RULE
CREATE SCHEMA
CREATE TRIGGER
CREATE VIEW
```

Ne mozes imati **DROP** pa onda posle toga **CREATE VIEW** jer

```
Msg 111, Level 15, State 1, Line 113
'CREATE VIEW' must be the first statement in a query batch.
```

Postavi ih u separate batches

### Batch as a unit of resolution

Batch is a unit of resolution(aka binding). To znaci da proveravanje postojanja objekta i kolona se desava na nivou batcha. Kada pravis schema changes nad objektom i pokusas da manipulises object data u istom batchu, sql server mozda nece biti svestan tih schema promena i mozda ce i failovati data manipulation statement.

```
ALTER TABLE dbo.T1 ADD col2 INT;
SELECT col1, col2 FROM dbo.T1;
```

Even though the code might seem to be perfectly valid, the batch fails during the resolution phase with the following error:

```
Msg 207, Level 16, State 1, Line 130
Invalid column name 'col2'.
```

Resenje: razdvoji DDL od DML querya sa “GO”

## The GO n Option

GO command se koristi od strane sql server client tools poput ssms da se oznaci kraj batcha.

Ta komanda podrzava i argument "koliko zelis da ponovis ovaj batch"

```
INSERT INTO dbo.T1 DEFAULT VALUES;
```

```
GO 100
```

```
SELECT * FROM dbo.T1;
```

➔ videces 100 inserta

## FLOW Elements

Sluze da umeris "flow" tvog koda.

### IF/ELSE flow statement

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
```

```
    PRINT 'Danas je zadnji dan u godini'
```

```
ELSE
```

```
    PRINT 'Nije zadnji dan u godini'
```

Ako je predicate **FALSE/UNKNOWN** onda ce ici u "ELSE" block. Ako i samo ako je **TRUE**, onda ide u **IF** block.

Ako moras da izvrsis vise od jednog statementa u IF/ELSE section, onda moras koristiti statement block koji se oznacavaju sa **BEGIN & END**.

```
IF DAY(SYSDATETIME()) = 1
```

```
BEGIN
```

```
...
```

```
END;
```

```
ELSE
```

```
BEGIN
```

```
...
```

```
END;
```

### WHILE flow element

```
DECLARE @i AS INT = 1;
```

```
WHILE @i <= 10
```

```
BEGIN
```

```
    PRINT @i;
```

```
SET @i = @i + 1;  
END;
```

T-sql daje WHILE **element** koji executeuje statement/block statement dokle god je predicate TRUE. Kad je false/unknown, onda se terminateje.

Uz **BREAK** command mozes izaci iz while loop-a.

```
...  
IF @i = 6  
BREAK;
```

Ako zelis da “preskocis izvrsavanje za neki uslov u **WHILE** statementu, onda mozes koristiti **CONTINUE** command

```
...  
IF @i = 6  
CONTINUE;  
...
```

## Cursors

Query sa ORDER BY clause vraca “cursor” – nerelacioni result sa garancijom orderinga izmedju redova.

Kad vracas bez order by onda je to samo set/multi set. Sql & T-sql takodje podrzavaju object koji se naziva *cursor* kog mozes koristiti da procesiras rows iz resultata query-a one at a time in a requested order.

Ugl savet je svakako da koristis standardne set based metode filtriranja/manipulacije umesto cursora jer cursor ima performance overhead(many times slower) i idu protiv relacionog modela.

Takodje cursori zahtevaju imperativni pristup.

Naravno cursori imaju exceptions kada su korisni, npr ako moras da odradis neki administrativni zadatak za neku tabelu/view nad svakim indexom ili kad ti set based operacija jako lose performuje i ne mozes vise da ga tuneujes(retki slucajеви).

Rad sa kurzorima ukljucuje:

1. Declaring the cursor based on a query
2. Open the cursor
3. Fetch attribute values from the first cursor record into variables

4. Dok nisi dosao do kraja kursora, loopuj kroz curosr records.(while @@fetch\_status = 0) U svakoj iteraciji izvrsi neophodni processing i onda fetchuj attribures iz sledeceg row-a unutar tih variables.

5. Close the cursor

6. Deallocate the cursor

Naredni kod ce koriscenjem curosra izracunati running total quantity za svakog customera i mesec iz sales.custorders

-- supress messages indicating how many rows were affected

SET NOCOUNT ON;

DECLARE @Result AS TABLE

(custid INT, ordermonth DATE, qty INT, runqty INT, PRIMARY KEY(custid, ordermonth));

DECLARE @custid AS INT, @prvcustid AS INT, @ordermonth AS DATE, @qty AS INT,  
@runqty AS INT;

-- declare cursor based on a query (fast\_forward -> read only, forward only)

DECLARE C CURSOR FAST\_FORWARD FOR

SELECT custid, ordermonth, qty FROM Sales.CustOrders

ORDER BY custid, ordermonth

-- open it

OPEN C;

--fetch values

FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;

-- init variables

SELECT @prvcustid = @custid, @runqty = 0;

-- loop over

WHILE @@FETCH\_STATUS = 0

BEGIN

IF @custid <> @prvcustid

SELECT @prvcustid = @custid, @runqty = 0;

SET @runqty = @runqty + @qty

INSERT INTO @Result VALUES(@custid, @ordermonth, @qty, @runqty);

FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;

END;

```

CLOSE C;

DEALLOCATE C;

SET NOCOUNT OFF;

SELECT custid,
CONVERT(VARCHAR(7), ordermonth, 121) as ordermonth,
qty,
runqty
FROM @Result
ORDER BY custid, ordermonth;

```

custid	ordermonth	qty	runqty
1	2021-08	38	38
1	2021-10	41	79
1	2022-01	17	96
1	2022-03	18	114
1	2022-04	60	174
2	2020-09	6	6
2	2021-08	18	24
2	2021-11	10	34
2	2022-03	29	63
3	2020-11	24	24
3	2021-04	30	54
3	2021-05	80	134
3	2021-06	83	217
3	2021-09	102	319
3	2022-01	40	359
...			
89	2020-07	80	80
89	2020-11	105	185
89	2021-03	142	327
89	2021-04	59	386

Ovako nesto mozes i sa window function

```

SELECT custid, ordermonth, qty, SUM(qty) OVER(PARTITION BY custid ORDER BY
ordermonth ROWS UNBOUNDED PRECEDING) AS runqty FROM Sales.CustOrders ORDER
BY custid, ordermonth;

```

## Temporary tables

Privremene tabele služe za privremeno skladistetnje podataka(ko bi rekao).

Recimo da ti je neophodno da podaci budu prisutni samo u trenutnoj sesiji ili batchu, onda bi to koristio ili ako nemas permisije da kreiras permanent tabelu u user database.

Postoje x3 vste privremenih tabele:

- local temporary tables
- global temporary tables,
- table variables

Sve one se kreiraju u *tempdb* bazi

### Local temporary tables

Local temporary tables se kreiraju tako sto ih imenujes sa tarabom(##) kao prefix poput #T1.

Local temporary tabela je vidljiva samo sesiji koja ju je kreirala unutar scope-a koji ju je kreirao i svim "pod scopeovima" to creating levela. To znaci da ako imas x4 stored procedure 1,2,3 & 4. U stored procedure 2 se kreira temp tabela. SP2 poziva SP3 & SP4. Temp dabela je dostupna u SP2, SP3 & SP4, ali ne i SP1. Pri zavrsetku SP2 se unistava i local temp table.

Ako se temp table kreira u ad-hoc batch u outermost nesting level of the session(kad je @@NESTLEVEL = 0), vidljiva je svim subsequent batchovima i unistava se onda kada creating session disconnects.

Mogu se kreirati x2 temp table sa istim imenom jer sql server dodaje suffix na table name pa je naziv unique u tempdb. Tebe kao developeru to ne interesuje, ali eto sad znaš.

Scenario:

- imas skupu proceduru koja se izvrsava i zelis da zapamti rezultat
- ako storeujes neke intermediate results privremeno

```
CREATE TABLE #MyOrderTotalsByYear(  
orderyear INT NOT NULL PRIMARY KEY,  
qty INT NOT NULL);
```

```
INSERT INTO #MyOrderTotalsByYear(orderyear, qty) SELECT ...
```

```
SELECT * FROM #MyOrderTotalsByYear...
```

## Global temporary tables

Vidljiva je svim drugim sesijama, destroy se automatski kada se creating session disconnectuje i nema vise aktivnih referenci na tabelu.

Prefix sa # tarabe

CREATE TABLE ##T1(...)

Korisno je za kreiranje tabele gde mozes da delis podatke sa drugima, ne trebaju ti posebne permisije i postoji full DDL & DML access nad njima od strane svih.

Ako zelis global temp table da bude kreirana svaki put kad se sql server startuje i ne zelis da pokusava da destrojuje automatski tu tabelu, moras kreirati table preko stored procedure koja je marked kao startup procedure

⇒ see “sp\_procoption”

## Table variables

Slicno sa local temporary variable. Deklarises ih sa DECLARE i onda kazes TABLE i definises tabelu. Ove tabele fizicki postoje u tempdb, a ne samo “in memory”.

Vidljive su samo creating sesiji, a ako se izvrsavaju u batchu onda samo u current batch-u.

**Razlika u poredjenju sa drugim temp tabelama je sto ako se transakcija rollbackuje, promene u temp tabeli su takodje rollbackovane do promene ka table variables nisu.**

Samo izmene napravljene od strane active statementa koje su failovale ili terminated pre completion su undone.

Postoje performance razlike, ali bitno ti je da znas da table variables su the way to go ako imas scenario gde treba da drzis mali volumen podataka(nekoliko redova), ali local temp tables su odgovor u suprotnom.

DECLARE @T1 TABLE(...)

SELECT/INSERT i slicno.

## Table types

Table type mozes koristiti da preserveujes table definition.

```
CREATE TYPE dbo.OrderTotalsByYear AS TABLE  
(orderyear INT NOT NULL PRIMARY KEY,  
qty INT NOT NULL);
```

```
DECLARE @T1 AS dbo.OrderTotalsByYear;
```

```
DROP TYPE IF EXISTS dbo.OrderTotalsByYear;
```

Sa tipizacijom mozes i lakse ubacivati "tipove" jelte u funkcije/procedure pa tako lakse transferovati podatke.

## Dynamic SQL

Dynamic sql je kada konstruisees batch t-sql koda kao character string i onda ga kasnije executeujes.

Postoje x2 nacina za izvrsavanje ovog koda. To je uz EXEC(EXECUTE) command i koriscenjem sp\_executesql stored procedure.

### EXEC command

Prihvata character string u zagradama kao input i onda ga izvrsi.

```
DECLARE @sql AS VARCHAR = 'PRINT "Kita";'  
EXEC(@sql);
```

### sp\_executesql stored procedure

Malo vise je secure i fleksibilnija u smislu da ima interface tj podrzava input i output parametre.

Podrzava samo unicode character strings kao input batch of code.  
Performance je bolji od EXEC, stitis se od sql injection.

```
DECLARE @sql as NVARCHAR(100);  
SET @sql = N'SELECT orderid, custid, empid FROM Saes.Orders  
WHERE orderid = @orderid;';
```

```
EXEC sp_executesql  
@stmt = @sql,  
@params = N'@orderid AS INT',  
@orderid = 10248;
```

@stmt parameter je statement koji treba da se izvrsi  
@params parametar je deklaracija input i output parametara  
<poslednji> parametri su assignments input & output parameters razdvojeni zarezom

## Using pivot with dynamic sql

U static query moras unapred znati vrednosti po kojima ces odraditi "IN". Ovde ne moras jer ga mozes dinamicki generisati.

```
DECLARE @sql AS NVARCHAR(1000) = N'SELECT * FROM
(SELECT shipperid, YEAR(orderdate) as orderyear, freight FROM Sales.orders) AS d
PIVOT(SUM(freight FOR orderyear IN((SELECT STRING_AGG(QUOTENAME(orderyear), N',')
WITHIN GROUP(ORDER BY orderyear) FROM (SELECT DISTINCT(YEAR(orderdate)) as
orderyear FROM sales.Orders) as D))) as P;';

?????
EXEC sys.sp_executesql @stmt = @sql;
STRING_AGG funkcija konkatonira distinct order years itd...
```

## ROUTINES

Rutine su programabilni objekti koji enkapsuliraju kod da izracunaju rezultat ili izvrse aktivnosti.

Postoje:

- user defined functions
- stored procedures
- triggers

### User defined functions

Svrha im je da enkapsulira logiku koja nesto racuna i vrati rezultat.

Mozes scalar(single result vrate) i table valued(vracaju tabelu) UDFs imati.

Table udf mogu da se pojavljuju u FROM. Scalar bilo gde u query. Ne smeju imati nikakav side effect.

RAND & NEWID funkcije vracaju globally unique identifier i one imaju side effects. Zato ih ne smes u tvojim UDFs koristiti.

```
CREATE OR ALTER FUNCTION dbo.GetAge
(@birthdate as DATE,
@eventdate as DATE)
RETURNS INT
AS
BEGIN
RETURN
DATEDIFF(year, @birthdate, @eventdate)
CASE WHEN
100 * MONTH(@eventdate) + DAY(@eventdate) <
100*MONTH(@birthdate) + DAY(@birthdate) THEN 1 ELSE 0
```

```
END;  
END;  
GO
```

Funkcija moze imati vise od samo jednog RETURN clause-a u bodyu.

## Stored procedures

Rutine koje enkapsuliraju kod. Mogu imati input & output parametre, returnovati result sets querya i smeju da imaju side effects.

Mozes modifikovati podatke kroz stored procedure i apply schema changes.

Ako treba da promenis implementaciju stored procedure, primeni promenu sa ALTER PROC.

Sa SP mozes omoguciti funkcionalnosti userima bez da im dajes direct permissions, npr ako mu oduzmes permisiju da obriše podatke, mozes mu to "omoguciti" kroz SP pa na taj nacin zadrzis standarde.

Sav error handling code moze biti unutar procedure.

Stored procedures ti daju performance benefits jer se kompajliraju i cacheju execution plans. Takodje smanjuje se network traffic u poredjenju sa necim poput dynamic sql jer SP ne mora da procesira sav taj procedure code i prebacuje sa servera na server.

```
CREATE OR ALTER PROC Sales.GetCustomerOrders  
@custid AS INT,  
@fromdate AS DATETIME = '19000101',    will be defaulted to that value  
@todate AS DATETIME = '99991231',  
@numrows AS INT OUTPUT – output parameter  
AS  
SELECT...  
SET @numrows = @@rowcount;  
GO
```

I onda da je izvrsis:

```
DECLARE @rc AS INT;  
  
EXEC Sales.GetCustomerOrders  
@custid = 1,  
@numrows = @rc OUTPUT;  
SELECT @rc ass numrows;
```

## Triggers

Trigger je posebna stored procedura koja se ne izvrsava "eksplicitno" nego na osnovu nekog eventa.

Kada se taj event izvrsi, onda se i ona pozove. Eventovi mogu biti npr manipulation events(DML triggers) poput INSERT, data definition events(DDL triggers) poput CREATE TABLE.

Trigger se smatra za deo transakcije koja okida taj neki event. ROLLBACK TRAN unutar trigger's code ce rollbackovati sve izmene koje su sse dogodile i u triggeru i u transakciji koja je okinula taj neki event i pozvala trigger.

Triggers u SQL Server se okidaju po statementu, a ne po modified row.

## DML Triggers

Postoje x2 DML Triggera:

*after*

*instead of*

Ater trigger se okida nakon sto se event zavrsi. Definise se samo nad permanent tables.

Instead of trigger se okida umesto eventa. Moze biti definisan na permanent tables & views.

Unutar trigger code mozes pristupiti pseudo tabelama *inserted* & *deleted* koje sadrzi rows affected by the modification koja je prouzrokovala trigger.

*inserted* sadrzi nove redove u slucaju INSERT/UPDATE actiona.

*deleted* sadrzi stare redove u slucaju DELETE/UPDATE

U slucaju instead of triggera, *inserted* & *deleted* tables sadrze redove koji su trebali biti affected od strane modifikacije.

Primer za after trigger kod audita.

```
CREATE OR ALTER TRIGGER trg_T1_insert_audit ON dbo.T1 AFTER INSERT
```

```
AS
```

```
SET NOCOUNT ON;
```

```
INSERT INTO dbo.T1_Audit(keycol, datacol)
```

```
SELECT keycol, datacol FROM inserted;
```

```
GO
```

audit_lsn	dt	login_name	keycol	datacol
1	2022-02-12 09:04:27.713	SHIRE\Gandalf	10	a
2	2022-02-12 09:04:27.733	SHIRE\Gandalf	30	x
3	2022-02-12 09:04:27.733	SHIRE\Gandalf	20	g

## DDL Triggers

Auditing, policy enforcement & cache management.

U standardnom mssql server(box offer) mozes ih kreirati u x2 scopea:

database scope

server scope

Azure sql supports only database triggers so far.

Unutar triggera obtainujes info o eventu koji je prouzrokovao trigger sa funkcijom **EVENTDATA** gde dobijes info kao XML instance.

XQuery expressionom mozes ekstraktovati event atribut poput vreme/event type/login name

DDL\_DATABASE\_LEVEL\_EVENTS – svi DDL eventi na db levelu.

```
CREATE OR ALTER TRIGGER trg_audit_ddl_events ON DATABASE FOR
DDL_DATABASE_LEVEL_EVENTS
AS
SET NOCOUNT
DECLARE @ eventdata AS XML = eventdata();
INSERT INTO dbo.AuditDDLEvents(...)
VALUES(
    @eventdata.value('(/EVENT_INSTANCE/PostTime)[1]', 'VARCHAR(23)'),
    @eventdata.value('(/EVENT_INSTANCE/EventType)[1]', 'sysname'),
    @eventdata.value('(/EVENT_INSTANCE/LoginName)[1]', 'sysname'),
    @eventdata.value('(/EVENT_INSTANCE/SchemaName)[1]', 'sysname'),
    @eventdata.value('(/EVENT_INSTANCE/ObjectName)[1]', 'sysname'),
    @eventdata.value('(/EVENT_INSTANCE/TargetObjectName)[1]', 'sysname'),
    @eventdata);
GO
```

XQuery expression **.value** method iz event info + XML instance

## Error handling

TRY/CATCH koji se pise sa:

```
BEGIN TRY
... TSQL Code
END TRY
BEGIN CATCH
---
END CATCH
```

Ako ima errora, onda u catch. Npr Print 10/0 ce ti rezultovati u erroru tj division by zero. Code execution se prekida se u momentu kad se taj error pojavi.

Da uhvatis informacije o gresci/exceptionu/erroru koristis funkcije.

```
ERROR_NUMBER – dobijes integer o broju greske
ERROR_MESSAGE – dobijes error message text
ERROR_SEVERITY & ERROR_STATE – severity & state
ERROR_LINE – linija koda u kojoj se error desio
ERROR_PROCEDURE ime procedure u kojoj se error desio; NULL ako nije u proc
```

Da bi queryovao/dobio listu error number & messages koristis **sys.messages catalog view**.

Primer Catch blocka

```
END TRY
BEGIN CATCH

IF ERROR_NUMBER() = 2627
BEGIN
    PRINT '      Handling PK violation...';
END;
ELSE IF ERROR_NUMBER() = 547
BEGIN
    PRINT '      Handling CHECK/FK constraint violation...';
END;
ELSE IF ERROR_NUMBER() = 515
BEGIN
    PRINT '      Handling NULL violation...';
END;
ELSE IF ERROR_NUMBER() = 245
BEGIN
    PRINT '      Handling conversion error...';
END;
ELSE
BEGIN
    PRINT 'Re-throwing error...';
    THROW;
END;
```

Takav code moze da se enkapsulira i u stored procedure i reuseuje kasnije

```
CREATE OR ALTER PROC dbo.ErrInsertHandler
AS
SET NOCOUNT ON;

IF ERROR_NUMBER() = 2627
BEGIN
    PRINT 'Handling PK violation...';
END;
ELSE IF ERROR_NUMBER() = 547
BEGIN
    PRINT 'Handling CHECK/FK constraint violation...';
END;
ELSE IF ERROR_NUMBER() = 515
BEGIN
    PRINT 'Handling NULL violation...';
END;
ELSE IF ERROR_NUMBER() = 245
BEGIN
    PRINT 'Handling conversion error...';
END;

PRINT 'Error Number : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT 'Error Message : ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT 'Error State   : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT 'Error Line    : ' + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT 'Error Proc    : ' + COALESCE(ERROR_PROCEDURE(), 'Not within proc');

GO;
```

I onda u tvojim buducim catch blokovima

```
BEGIN CATCH

IF ERROR_NUMBER() IN (2627, 547, 515, 245)
EXEC dbo.ErrInsertHandler
ELSE
THROW;

END CATCH;
```